UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uma Arquitetura de Rede Programável para Redes Orientadas à Informação com Replicação de Conteúdo em Nuvens Privadas

Erick Barros Nascimento

São Cristóvão – Sergipe

2018

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Erick Barros Nascimento

Uma Arquitetura de Rede Programável para Redes Orientadas à Informação com Replicação de Conteúdo em Nuvens Privadas

Proposta de dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como requisito para a obtenção do título de mestre em Ciência da Computação.

Orientador(a): Prof. Dr. Douglas Dyllon Jeronimo de Macedo
Coorientador(a): Prof. Dr. Edward David Moreno

São Cristóvão – Sergipe

2018

Erick Barros Nascimento

Uma Arquitetura de Rede Programável para Redes Orientadas à Informação com Replicação de Conteúdo em Nuvens Privadas/ Erick Barros Nascimento. – São Cristóvão – Sergipe, 2018-

153 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Douglas Dyllon Jeronimo de Macedo

Dissertação de Mestrado – UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO, 2018.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

CDU 02:141:005.7

Erick Barros Nascimento

Uma Arquitetura de Rede Programável para Redes Orientadas à Informação com Replicação de Conteúdo em Nuvens Privadas

Proposta de dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como requisito para a obtenção do título de mestre em Ciência da Computação.

Trabalho aprovado. São Cristóvão – Sergipe, 30 de Maio de 2018:

Prof. Dr. Douglas Dyllon Jeronimo de Macedo
Orientador

Prof. Dr. Edward David Moreno Ordoñez
Coorientador

Prof. Dr. Luis Carlos Erpen de Bona
Universidade Federal do Paraná (UFPR)

Prof. Dr. Tarcísio da Rocha
Universidade Federal de Sergipe (UFS)

São Cristóvão – Sergipe
2018

Eu dedico essa dissertação ao meu Pai (in memoriam), toda minha família, amigos e professores, que através da graça de Deus concedida por eles, consegui me sustentar até aqui.

Agradecimentos

A Deus, Nosso Senhor, para todo o sempre em nome do Espírito Santo, amém. Por ter me sustentado nessa árdua caminhada. Pela proteção nas cansativas viagens. Por me fazer não desistir nos momentos de desespero. Eu te agradeço ó Pai!

Ao meu Pai (*in memorian*), porque senti por diversas vezes tua presença. Sei que estais feliz pelo que me tornei. E sei que está comigo em todos os momentos.

A minha mãe, pois eu sei que em todas as suas orações pediu a Deus por mim. Sofreu junto comigo quando eu chegava em sua casa triste por achar que não daria certo. Mainha, amo muito você.

A minha esposa Malú e minha filha Lavínia. Meu alívio foi não ter colocado o curso em sua frente filha, isso traz orgulho a mim mesmo. Se eu faltei com alguma coisa, peço perdão, sem vocês, a vida perde o sentido.

Aos meus queridos amigos Alan Amorin e Othon Stuart, por me abrigar em seu aconchegante lar, me dando total liberdade de ir e vir em suas casas. Caras! Não sei como agradecer, contem comigo sempre e para sempre.

Aos meus amigos do Mestrado: Cícero, Thauanne, Breno, Marianne e Davy, pelos debates e reuniões no laboratório do curso. Vocês são muito tops! Sentirei saudades.

Ao meu orientador, professor Dr. Douglas Dyllon Jeronimo de Macedo, pela paciência e maestria para me orientar. Sabendo conduzir o aluno, "como poucos"eu acredito, direcionando efetivamente para construção de uma parceria produtiva. Não tenho palavras para te agradecer, você é um excelente professor, obrigado de coração.

Ao meu coorientador, professor Dr. Edward David Moreno, pelas ótimas aulas na época das disciplinas, me alertando sobre o meu modo de me apresentar.

Ao professor, Dr. Luis Carlos Erpen de Bona (UFPR), que mesmo a distância, e sem conhecê-lo pessoalmente, avaliou de forma precisa minha pesquisa, obrigado.

Ao professor Dr. Methanias Colaço, porque em todos os e-mails citou sobre o potencial da minha pesquisa, obrigado, você é um grande mestre!

Ao meu colega de mestrado, Bráulio Lívio, pelas inúmeras conversas em relação ao nosso desenvolvimento no curso. Sempre incentivando um ao outro. Valeu parceiro! Conte comigo sempre.

A professora Dr^{a} Tharsia Costa (IFBA), pela paciência para tirar minhas dúvidas e acreditar na minha pesquisa. Obrigado minha querida!

Ao Instituto Federal da Bahia (IFBA) Câmpus Paulo Afonso, em nome de Prof. Arleno (Diretor), Prof^{a} Esdriane (Diretora de Ensino) e Lídia Sandes (Gabinete), obrigado!

Por fim, ao meu professor e colega de trabalho, Msc. Ricardo Porto, pela paciência junto ao curso de Sistemas (FASETE), entendendo minha rotina de vida. Obrigado! Conte comigo.

Na minha vida nem tudo acontece,
Mas quanto mais agente rala, mais agente cresce
Com a cabeça erguida e mantendo a fé em Deus
O seu dia mais feliz vai ser o mesmo que o meu
A vida me ensinou a nunca desistir
Nem ganhar, nem perder, mas procurar evoluir
Podem me tirar tudo que tenho
Só não podem me tirar as coisas boas
Que eu já fiz pra quem eu amo
E eu vou que vou
História, nossas histórias
Dias de luta, dias de glória
Hoje estou feliz
Sintonia
Telepatia
Comunicação pelo córtex
Boom
Bye, bye
(Chorão)

Resumo

A Internet lidera o modo como as informações, serviços e conteúdos são disponibilizados em formato digital. O acesso a esses conteúdos pode ser possível de qualquer lugar, pelos mais variados dispositivos. Entretanto, a configurabilidade da arquitetura de redes convencional é dificultada pela diversidade de sistemas embarcados presentes nos ativos de rede, onde as informações que consumem seus recursos não possuem um gerenciamento efetivo, ao mesmo tempo que, alguns problemas de tráfego não são detectados com facilidade pelos sistemas e nem pelas equipes de infraestrutura, como a duplicação do tráfego de requisições HTTP no enlace.

Esta pesquisa realiza um estudo experimental com proposta de arquitetura descentralizada para redes de conteúdo. A arquitetura centraliza todo o gerenciamento desacoplando o controle dos concentradores da rede, permitindo a aplicação das configurações *on-the-fly*. Com o advento das redes definidas por *software* (do inglês, *Software Defined Networking (SDN)*), surgem novas possibilidades para o gerenciamento de redes, promovendo novos contextos para as redes cêntricas de informação (do inglês, *Information Centric Network (ICN)*). Essas possibilidades podem ser alcançadas por meio da programabilidade da rede com extensão remota do plano de dados para nuvem.

Este trabalho apresenta e valida um protótipo de redes de conteúdo, disponibilizado em um domínio estabelecido e replicado entre os mesmos, onde cada *endpoint* executa um microcache para otimização do tempo de requisição e resposta. Além disso, uma cópia minimizada desse *backend* está posto na nuvem em um servidor privado (do inglês, *Virtual Private Server (VPS)*) permitindo a alta disponibilidade do ambiente. O sistema será composto por dois proxies reversos capazes de assumir o controle do plano de dados sem percepção para o usuário, mantendo alta performance e otimização através do algoritmo de balanceamento de carga Least_Conn.

A validação ocorreu por meio de três cenários. No primeiro, foram avaliados aspectos inerentes ao funcionamento de uma rede de computadores, logo, latência, perda e vazão foram explorados. No segundo cenário, foram avaliados aspectos acerca do tunelamento para a nuvem com requisição direcional e bidirecional obtendo atraso e perda de pacotes. Por fim, aspectos de replicação, cacheamento, banda de contenção, otimização, banda alcançável, Cache Hit Ratio (CHR), Tempo Médio entre Falhas (MTBF) e Tempo Médio para Reparos (MTTR) foram apresentados, alcançando bons resultados para ICNs com SDN. O trabalho é concluído apontando para o estudo da latência de controladores SDN devido a variações no sistema, mas que não impactaram significativamente nos objetivos da pesquisa.

Palavras-chave: Computação em Nuvem, ICN, SDN, Sistemas Distribuídos, Redes Programáveis, Cache, Proxy Reverso, Controlador.

Abstract

Internet leads with the information, services and content are made available in digital format. Access to such content can be possible from anywhere, by the most varied devices. However, the configurability of the conventional network architecture is hampered by the diversity of embedded systems deployed in the network assets, where the information that use its resources does not have an effective management, while specific problems are not easily detected by the systems and infrastructure teams, for instance, duplicated HTTP traffic requests in the network segment.

This research conducts an experimental study with proposal of a decentralized architecture for content networks. The architecture centralizes all management by decoupling the concentrators control of the network, allowing the application of the configurations on the fly. With the advent of Software Defined Networking (SDN), new possibilities for network management are emerging for new contexts for the Information-centric Network (ICN). These possibilities can be achieved through network programming with remote extension of the data plane to the cloud.

This work presents and validates a prototype of content networks, made available in an established domain and replicated between them, where each endpoint perform a microcache to optimized the request and response time. In addition, a minimized copy of this backend is placed on the cloud in a Virtual Private Server (VPS)) enabling high availability of the environment. System has two reverse proxies capable of taken control of the data plane without perception to the user, keeping high performance and optimization through the Least_Conn load balancing algorithm.

Validation took place through three scenarios. In the first, it was evaluated aspects inherent to the operation of a computer network, therefore, latency, packets loss and flow rate. In the second scenario, aspects were evaluated regarding the tunneling for the cloud with directional and bidirectional request to obtaining delay and packets loss. Finally, aspects of replication, caching, contention band, optimization, reachable bandwidth, Cache Hit Ratio (CHR), Mean Time Between Failure (MTBF) and Mean Time To Repair (MTTR) were presented, achieving good results with ICN and SDN. The work finalize pointing to study of controller latency due to variations in the system, but this did not impact the research objectives.

Keywords: Cloud Computing, ICN, SDN, Distributed Systems, Programmable Networks, Cache, Reverse Proxy, Controller.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

API	*Application Program Interface*
ARP	*Address Resolution Protocol*
BDAAS	*Big Data as a Service*
BGP	*Border Gateway Protocol*
BIND	*Berkeley Internet Name Daemon*
CCN	*Content Centric Network*
DBAAS	*DataBase as a Service*
DFD	*Diagrama de Fluxo de Dados*
DNS	*Domain Name Sytem*
DONA	*Data Oriented Network Architecture*
GRE	*Generic Routing Encapsulation*
HTTP	*Hyper Text Transfer Protocol*
ICMP	*Internet Control Message Protocol*
ICN	*Information-Centric Network*
IOT	*Internet of Things*
IP	*Internet Protocol*
IAAS	*Infrastructure as a Service*
IETF	*Internet Engineering Task Force*
LAN	*Local Area Network*
LVS	*Linux Virtual Server*
MAC	*Media Access Control*
NIST	*National Institute Standard and Technology*
NRS	*Name Resolution Service*
NDO	*Named Data Objects*

NOS	*Network Operation System*
ODL	*OpenDaylight*
OF	*Open Flow*
ONF	*Open Networking Foudation*
ONOS	*Open Network Operating System*
OSPF	*Open Shortest Path First*
OVSDB	*Open vSwitch Database Management*
QOS	*Quality of Service*
PAAS	*Plataform as a Service*
PSIRP	*Publish Subscribe Internet Routing Paradigm*
PURSUIT	*Publish Subscribe Internet Technology*
RIP	*Routing Information Protocol*
RTT	*Round Trip Time*
SAAS	*Software as a Service*
SAIL	*Scalable and Adaptive Internet Solutions*
SDN	*Software Defined Networking*
SDWAN	*Software Defined Wide Area Network*
SGDB	*Sistema de Gerenciamento de Banco de Dados*
SLR	*Systematic Literature Review*
SNMP	*Simple Network Management Protocol*
SSH	*Secure Shell*
SAS	*Storage Area Network*
TCP	*Transmission Control Protocol*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
VCNI	*Cisco Virtual Networking Index*

VLAN	*Virtual Local Area Network*
VM	*Virtual Machine*
VMM	*Virtual Machine Monitor*
VPS	*Virtual Private Server*
VRRP	*Virtual Router Redundancy Protocol*
VxLAN	*Virtual Extensible LAN*
WAN	*Wide Area Network*

Sumário

1 Introdução

A arquitetura de redes de computadores poderá passar por mudanças em um futuro próximo, cujo o objetivo é facilitar a configurabilidade da rede de maneira menos impactante para os operadores de rede. Essa configurabilidade muitas vezes é dificultada pela diversidade de sistemas embarcados presentes nos ativos de rede (*switches, routers e middleboxes*), esses sistemas são de extrema importância para sustentação da infraestrutura topológica de redes num contexto geral. Além disso, as informações que circulam na infraestrutura consumindo seus recursos não possuem um gerenciamento efetivo, podendo acarretar problemas de indisponibilidade de serviços, ao mesmo tempo que, alguns problemas específicos não são detectados com facilidade pelas equipes de infraestrutura. Dessa forma, um requisito indispensável é que as redes sejam dinâmicas e tenham atualização imediata no momento da operação (LI et al., 2017).

De acordo com a (CISCO, 2014), estima-se que até 2021 teremos cerca de 4,6 bilhões de usuários de Internet, com 27,1 bilhões de dispositivos conectados e conexões, sendo que 80% desse tráfego será destinado a entrega de conteúdo em formato de vídeo. Ainda que tenhamos conexões de alta velocidade que suportem o tráfego de uma grande quantidade de dispositivos, o modelo convencional ainda depende da comunicação ponto a ponto para estabelecer o canal de comunicação entre dois sistemas finais. Novas arquiteturas surgem na tentativa de otimizar o modelo baseado em IP projetando formatos de rede para conteúdo distribuído que flexibilizem o escalonamento das requisições e aumente a eficiência do sistema.

Além do desafio da busca por gerenciamento centralizado, o formato de disponibilização de conteúdo fornecido por meio de nós endereçados através de IPs, também tem uma forte tendência a ter seu formato encapsulado pelas redes de conteúdo, ou seja, as requisições de informação feitas a computadores que possuem endereçamento IP associados à resolução de nomes, são executados diretamente sob o conteúdo solicitado, e a rede de conteúdo se encarregará de entregar a informação atendendo a todos os requisitos que englobam os princípios fundamentais das redes de computadores. Para apoiar este último, as redes definas por *software* (do inglês, *Software Defined Networking, SDN*), trazem um gerenciamento de formato centralizado (sem dependência de *software* embarcado de fabricantes), e ainda orquestrado por aplicações executadas sobre o ambiente de rede, motivando desafios de infraestrutura e aplicações para sistemas distribuídos.

1.1 Contextualização

A Internet lidera o modo como as informações e os serviços são disponibilizados em formato digital. O acesso a esse conteúdo pode ser possível de qualquer lugar, pelos mais variados dispositivos, bastando apenas que esses dispositivos tenham uma conexão com a Internet. Entre-

tanto, o controle distribuído, a arquitetura de rede e os protocolos de transporte, são requisitos fundamentais para que a informação seja transferida digitalmente. Neste sentido, a infraestrutura convencional que suporta as conexões de rede, embora difundida e consolidada em qualquer ambiente (seja na indústria ou em outro segmento), ainda permanece muito complexa e com gerenciamento difícil (KREUTZ et al., 2015). Independentemente da verticalização imposta pelos fabricantes de ativos de rede, novas formas de estruturação desses ambientes estão surgindo, beneficiando a flexibilização de configuração desses ambientes pela programabilidade.

As redes de computadores convencionais, são o resultado de projetos de protocolos para transportar informação iniciados por volta da década de 70 (CARVALHO, 2006). Naquela época, acreditava-se que aquela infraestrutura de rede poderia ser suficiente para suportar todas os requisitos que a Internet necessitaria, mas a expansão da tecnologia e o aparecimento dos recursos disponibilizados de forma *on-line*, culminou com o aumento da utilização de recursos de rede, e com a necessidade pelo desenvolvimento de topologias flexíveis. Além disso, as redes foram consideradas estáticas, por que uma vez estabelecido uma topologia de rede, caso fossem necessárias alterações em sua estrutura nem sempre eram possíveis de forma flexível, causando alto impacto aos usuários finais e aos sistemas dependentes desse modelo de arquitetura (MORREALE; ANDERSON, 2015).

Um outro ponto, é que os servidores físicos que entregam serviços às infraestruturas conectadas a esses servidores passaram por mudanças significativas em termos de expansão e otimização do *hardware*. A chegada da virtualização mudou a forma de funcionamento desses equipamentos. Agora os servidores são dinâmicos, e são criados e movidos facilmente, e ainda são capazes de usar os recursos de rede da mesma forma dos servidores estáticos e físicos. Antes da chegada da virtualização em larga escala, as aplicações eram associadas a um único servidor que tinha local fixo na rede. Essas aplicações trocavam informações com outras aplicações que também ficavam armazenadas em servidores estáticos, configurados para disponibilizar uma única aplicação ou um único serviço mudando completamente a forma de disponibilização dos recursos na rede. No cenário atual, aplicações e os conteúdos podem ser distribuídos por meio de múltiplas máquinas virtuais, do inglês *virtual machines* (VMs). Cada VM pode trocar fluxo de dados com outras VMs. Servidores podem ser movidos de um local para o outro, flexibilizando o balanceamento de carga. Movimentação de aplicações através de fluxo de informações também são recursos explorados, onde a migração de servidores e aplicações criam novos desafios para os administradores de rede, incluindo esquemas de endereçamento, espaço de nomes baseados em função de segmentos, escalabilidade do sistema baseado em roteamento, etc (ONF, 2012).

Os sistemas que controlam os ativos de rede também estão passando por uma mudança que possivelmente trará benefícios positivos da mesma forma que a virtualização de servidores trouxe para os ambientes de redes e de *datacenter*. Nos concentradores atuais, o *data plane* das redes são integrados verticalmente, isso significa que o controle do sistema (*control plane*) e o plano de dados (*data plane*) são construídos na mesma arquitetura de *hardware* dos equipamentos. Isso reduz a flexibilidade de topologia e de gerência por conta das características individuais dos

diversos *software* dos fabricantes de equipamentos de redes. As SDNs possibilitam resolver os problemas de gerenciamento, opostamente ao *software* embarcado do concentradores (KREUTZ et al., 2015).

As arquiteturas de redes convencionais se tornaram limitadas quando são construídas em torno da comunicação ponto a ponto. Nesse tipo de comunicação a conexão é estabelecida entre um lado requisitante e um lado receptor e por meio desse canal de comunicação, sejam eles, *Simplex*, *Half-Duplex* ou *Full-Duplex*, com uma conexão persistente ou não persistente, um único caminho entre origem e destino passa a existir. Desde modo, caso ocorra uma nova conexão em direção à mesma informação que se pretenda obter, uma concorrência sob o mesmo canal de comunicação irá acontecer, ou seja, a mesma informação será trocada duplicadamente pelo mesmo segmento de rede. Isso reforça uma problemática no aumento da latência da rede, na diminuição dos recursos de transferência de dados, perda de pacotes, e falhas de conexão.

A programabilidade poderá mitigar o problema da gerencia da rede causado pela verticalização dos equipamentos. A característica de centralização permitirá administrar a rede de um único ponto. Além desse último, a computação em nuvem trará a estrutura necessária de forma extensa e escalável para construção de novos mecanismos que suportem a flexibilização dos ambientes de rede, abrindo espaço para outras formas de acesso a conteúdos e utilização de sistemas em nuvem, como por exemplo, *Information-Centric Network* (ICN) e *Virtual Private Servers* (VPS).

O paradigma Information-Centric Network (ICN) traz o uso da informação através de *caching* na rede, comunicação multi-parte através de replicação e interação de modelos separando remetentes e destinatários. O objetivo do ICN é melhorar a distribuição da informação massiva (com conteúdo distribuído e móvel) para que não haja concorrência de requisição ao conteúdo que será transportado, mantendo cópias inteiras ou partes (*chunks*) distribuídas pela rede encapsulando as conexões fim a fim sendo imperceptíveis para os requisitantes. Esse tipo de abordagem já vem sendo estudado pela comunidade científica, com foco na resolução de nomes de conteúdo, tal como um *Domain Name System (DNS)*, e no roteamento de requisições baseado em conteúdos.

Neste trabalho é proposta uma arquitetura ICN (*Information-Centric Network*), baseada em SDN, com persistência dos dados em uma VPS privada. Com essa proposta espera-se contribuir para um dos desafios das redes programáveis (flexibilidade), contribuir para prover arquiteturas de conteúdo que funcionem sob TCP/IP e utilizar volumes de armazenamento descentralizado. Assim, é possível utilizar o formato SDN para suportar o uso de redes de conteúdo, sendo escalável e flexível, e ainda replicar seus arquivos (caches) em volumes lógicos na nuvem. A Figura 1 representa um *layout* de uma arquitetura de rede convencional. No lado esquerdo temos *switches* e roteadores interligados entre si e complexos protocolos de roteamento para construção do *data plane*. Uma vez estabelecido um *data plane*, as configurações são aplicadas sobre cada *hardware* individualmente, seja o protocolo de roteamento intra sistema autônomo (OSPF, RIP), ou um protocolo complexo na borda da rede (BGP). No lado direito, temos uma

rede programável gerenciada por uma unidade centralizada. Essa unidade centralizada é denominada controlador e possui um sistema operacional que controla esse *data plane*. Uma vez estabelecido um evento ou ação, o controlador fará o *deploy* entre os *switches* de uma única vez (manualmente), ou através de gatilhos previamente programados fazendo um *deploy* dinâmico.

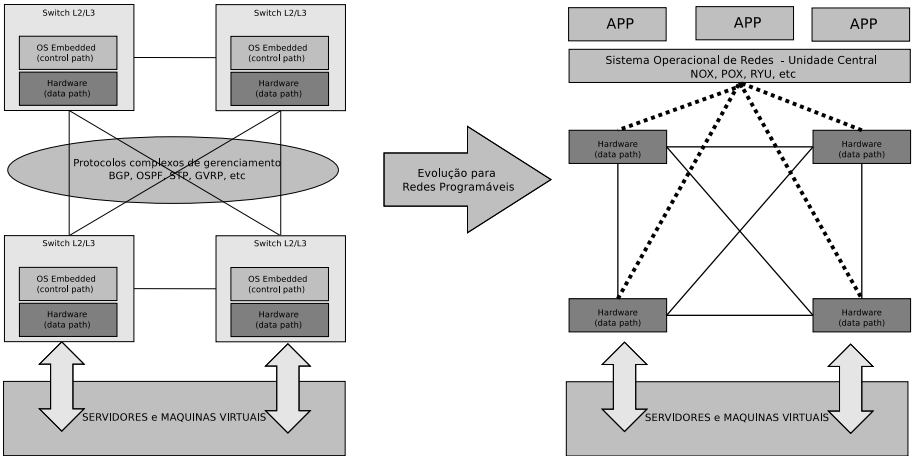


Figura 1 – Rede convencional evoluída para o paradigma programável.

1.2 Desafios e oportunidades

Os paradigmas Sofware Defined Networking (SDN) e Information Centric Network (ICN) têm sido amplamente discutidos, chamando atenção para esses novos modelos de infraestrutura de redes e aumentando o interesse das comunidades de pesquisa. Diante de suas características programáveis, os projetos SDN e ICN crescem paralelamente. Além de serem arquiteturas projetadas para flexibilizar domínios de rede, espera-se mitigar falhas imperceptíveis devido ao seu modelo de controle centralizado, entregando conteúdo através de uma estrutura de rede escalável e com gerenciamento descomplicado (GAO et al., 2016). Além dos benefícios que acompanham essas duas abordagens, os estudos apontam para um futuro certo na forma de transmissão de informação, e na forma como os conteúdos serão transportados por essas redes até os sistemas finais. Esses sistemas deverão atender a todos os requisitos já existentes nos ambientes convencionais tais como: controle de tráfego, correção de erros, políticas de segurança da informação, etc.

Em um primeiro momento, a Internet teve seu foco somente na conexão entre dois *hosts*. Além disso, quando solicitamos uma informação oriunda da Internet, precisamos nos preocupar com a fonte dessa informação para poder garantir que o recurso solicitado esteja sendo entregue através de uma fonte confiável. No entanto, o mapeamento da localização de armazenamento desses dados traz um desafio na configuração das redes de conteúdo se compararmos com os princípios da segurança de redes e aplicações: confiabilidade, disponibilidade e integridade (SON et al., 2016), além do desafio para SDN de como alinhar alta disponibilidade, segurança, flexibilidade e escalabilidade, e ainda mais resolver problemas de latência do controlador, tipo

de armazenamento para o controlador, comunicação entre controladores. Todos esses pontos observados levaram à motivação por este trabalho de pesquisa.

Sendo assim, foi idealizada uma arquitetura que pudesse prover conteúdo confiável replicado através da rede, possibilitando o armazenamento de conteúdos confiáveis por entre nodos de uma rede, permitindo que requisições individuais ao sistema sejam controladas individualmente, sem que o conteúdo seja enviado e recebido no mesmo segmento da rede repetidamente, forçando um aumento da banda passante e gerando *overhead* desnecessário devido ao crescente número de requisições a serviços em sistemas (TRAJANO; FERNANDEZ, 2016). Outro aspecto importante, é que o conteúdo que será transportado por um sistema autônomo SDN poderá não possuir local centralizado para armazenamento desses arquivos, evidenciando uma rede de informações sem redundância. Logo, alguns dos desafios que envolvem SDN/ICN são relacionados abaixo:

1. *Como aproveitar características SDN para suportar uma rede de conteúdo?*
   A programabilidade dos ativos de rede e a centralização do gerenciamento em controladores, têm o potencial de modificar a forma que os dados poderão ser transportados em comparação aos ambientes convencionais. Eles trazem novas possibilidades na implementação dinâmica de fluxos e na simplificação do mecanismo de controle.

2. *Como uma rede SDN poderá facilitar o gerenciamento do data plane de uma rede ICN?*
   O gerenciamento centralizado trará maior flexibilidade no controle, já que o mesmo estará em um único ponto ao alcance do administrador da rede, e não em pontos descentralizados geograficamente, onde cada concentrador possui seu próprio sistema embarcado.

3. *Como construir uma rede SDN/ICN para utilizar cópias em nuvem?*
   Com o uso de recursos computacionais orquestrados em *clouds*, a implementação dinâmica se faz cada vez mais necessária, disponibilizando meios para atender o crescimento exponencial de dispositivos conectados ao ambiente, tornando necessário que a rede de conteúdo tenha um lugar externo à LAN para armazenar os dados(caches) que circulam internamente.

Este trabalho de pesquisa aborda os três desafios enumerados acima com intenção de tratar aspectos relacionados à distribuição de dados para redução da vazão da rede e interligação de serviços em nuvem. A arquitetura da pesquisa irá utilizar as redes SDN por conta da centralização do gerenciamento, flexibilidade por ser programável, e por ser adaptável com outras tecnologias.

1.3 Objetivos

1.3.1 Objetivo Geral

Avaliar uma arquitetura de rede de conteúdo *Information-Centric Network* (ICN) suportada através de *Software Defined Network* (SDN), com o propósito de replicar o conteúdo em cache através da rede para minimização do tráfego, evitando a concorrência de requisições no mesmo segmento mantendo cópias confiáveis utilizando recursos em nuvens computacionais.

1.3.2 Objetivos Específicos

- Desenvolver um *datacenter* programável que possa ser utilizado em ambientes para prover a entrega de conteúdos entre requisições ao sistema. A topologia da rede deverá ser capaz de manter conexões em nível de Gbps desde o nível de LAN, até mesmo nível de WAN.
- Identificar, analisar e comparar, estudos sobre *software defined network* com contexto aplicado a *information-centric network* e as formas de armazenamento de dados em nós requisitantes.
- Implementar uma topologia de rede que possa fazer o mapeamento de fluxos requisitantes dinamicamente com intervenção somente na inicialização da rede, executando o *deploy* quando houver uma requisição inicial.
- Desenvolver um microsserviço que seja capaz de disponibilizar conteúdo dinamicamente, e que esse conteúdo seja identificado através de resolução de nomes e endereços.
- Implementar uma conexão externa (em nuvem) que possua *data plane* próprio e que esse mesmo esteja conectado ao *data plane* local sem percepção dos requisitantes do sistema. Essa conexão deverá se conectar remotamente e ainda operar sob a mesma faixa de rede.
- Implementar o sistema de armazenamento em nuvem. Esse armazenamento deverá guardar dados localmente bem como remotamente. O acesso ao conteúdo deverá ser imperceptível aos clientes, sem a necessidade de obtenção da origem dos dados.

1.4 Contribuições

As contribuições que esta pesquisa apresenta são enumeradas na sequência:

1. Abordagem para otimização do consumo da banda da rede pela distribuição do conteúdo dinamicamente, melhorando a entrega dos serviços e garantindo integridade das informações por cópia segura descentralizada.

2. Uma Arquitetura que utiliza caches replicados para otimizar a concorrência pela duplicação de dados transferidos no mesmo segmento, reduzindo o tempo de processamento total da requisição pela distribuição de requisições dinamicamente pelo domínio que opera a rede.

3. Um mecanismo de tunelamento para nuvem, com baixo *overhead* e alta velocidade, mantendo ambos os lados do *data plane* heterogêneos e com baixa latência.

4. Disponibilização de um *script* proposto para várias topologias.

5. Disponibilização do Código fonte da Rede Programável.

6. Disponibilização do código do controlador e dos microsserviços instanciados por ele, com comandos para criar túneis entre controladores totalmente funcionais.

7. Disponibilização dos arquivos de configuração do balanceador de carga e dos proxies.

1.5 Estrutura do trabalho

Este trabalho está estruturado em seis seções, e os demais capítulos são apresentados como segue:

O Capítulo 2 apresenta os conceitos teóricos necessários ao entendimento do trabalho, bem como a revisão sistemática e os trabalhos relacionados que ajudaram a embasar o desenvolvimento da pesquisa.

O Capítulo 3 apresenta a metodologia usada neste trabalho. O método utilizado é descrito, bem como as atividades que foram realizadas. Também serão apresentados todos os materiais e recursos utilizados para realização dos experimentos. Ao final serão detalhadas as métricas usadas para compor a validação.

O Capítulo 4 detalha a implementação do protótipo, apresentando todos os elementos necessários para sua composição. Serão descritos todas as configurações de *hardware* e *software* necessários para o funcionamento, bem como a infraestrutura da nuvem necessária para replicação.

No Capítulo 5 é apresentada a validação da arquitetura, descrevendo as medições realizadas e os resultados alcançados. Neste capítulo são discutidos os resultados obtidos e os gráficos são apresentados, mostrando de forma intuitiva a relação das métricas experimentadas.

Por fim, o Capítulo 6 apresenta a conclusão do trabalho, suas limitações, suas contribuições e a proposta para trabalhos futuros. Neste capítulo, os objetivos específicos são confrontados para verificar se a pesquisa irá direcionar para uma continuidade acerca da questão levantada no tema.

2 Fundamentação Teórica

A programabilidade das redes foi inicialmente desenvolvida por meio de trabalhos de pesquisa iniciados no ano de 2004 como parte da busca sobre um novo paradigma de gerenciamento de rede, que resultou em dois projetos distintos: o primeiro, um ambiente para controle de roteamento, trabalho feito na Universidade de Pricenton e Carnegie Mellon (CAESAR et al., 2005). O segundo, um trabalho sobre segurança de redes executado em paralelo como parte de um projeto chamado SANE Ethene da Universidade de Stanford, e da Universidade da Califórnia em Berkeley (AKONJANG, 2007). O projeto inicial foi desenvolvido por dois grupos diferentes, uma *startup* chamada Nicira, comprada pela VMWare, que criou o sistema operacional de rede NOX (escrito em C/C++) (GUDE et al., 2008), e a Nicira, trabalhando com grupos de pesquisa da Universidade de Stanford na criação da interface OpenFlow Switch (ONF, 2012).

Em 2011, com o amplo apoio da indústria, a Open Networking Foudation (ONF), firmou a adoção das redes SDN e obteve apoio de grandes fabricantes de *hardware*, tal como: Cisco, Juniper Networks, Hewllet-Packard, DELL, Broadcom e IBM. E também recebeu apoio de gigantes das comunicações como: Google, Verizon, Yahoo, Microsoft, Deutsche, Telekom, Fecebook e NTT (MORREALE; ANDERSON, 2015). A ONF é uma organização dedicada ao fomento e adoção do Software Defined Networking (SDN) através de padrões abertos de desenvolvimento. A ONF sintetiza SDN como: ”Uma nova abordagem para as redes de computadores, de modo que o controle está dissociado da função de transmissão de dados e é diretamente programável através de uma linguagem de programação de alto nível “(ONF, 2016, p. 1). O resultado é uma arquitetura extremamente dinâmica, controlável, de baixo custo, e adaptável, flexibilizando o gerenciamento dos operadores pela programação, automação e controles centralizados. É implementado sob um padrão de desenvolvimento aberto, permitindo que os administradores possam integrar uma rede programável através de uma ampla variedade de outras tecnologias (ONF, 2016).

2.1 Sofware Defined Networking (SDN)

A definição mais genérica para conceituar SDN, é que SDN é uma arquitetura de rede onde o encaminhamento dos segmentos no *data plane* é gerenciado por um *control plane* remoto desacoplado da forma embarcada dos concentradores de rede. A indústria define que qualquer equipamento de rede que envolva programabilidade é uma forma de SDN. Segundo Kreutz et al. (2015), SDN pode ser definido como uma arquitetura de rede contendo quatro pilares fundamentais:

1. O *control plane* e o *data plane* são desacoplados, ou seja, a funcionalidade de controle é removida dos concentradores de rede tornando-os encaminhadores de pacotes.

2. Decisões de encaminhamento são baseadas em fluxo (*flow-based*) ao invés de destino (*destination-based*) como nos equipamentos convencionais. Em SDN/Openflow, um fluxo é uma sequência de pacotes entre a origem e o destino abstraindo o encaminhamento de pacotes por meio do mapeamento do caminho origem e destino, permitindo a unificação do seu comportamento em diferentes equipamentos de rede (JAMJOOM; WILLIAMS; SHARMA, 2014).

3. O controle lógico (*software*) é movido para uma unidade externa denominado *controller* ou *Network Operation System* (NOS), que analogamente pode ser considerado o sistema operacional da rede. O NOS é um sistema cliente/servidor que fornece os recursos necessários para facilitar a programabilidade dos dispositivos que segmentam a rede em um ambiente centralizado, que por sua vez, encapsula a visualização do ambiente de rede.

4. A duas principais características das redes SDN são a centralização do controle e a programabilidade. A rede é programada na camada superior do NOS e interage com os dispositivos (programáveis) na camada subjacente do sistema, no *data plane*.

O encaminhamento de pacotes é uma das funções primárias de um *data plane*. Além da capacidade de programabilidade como um equipamento físico de rede, realiza funções de inspeção de pacotes (*payloads*) e processamento de requisições da rede, por exemplo, *caching*. Diante disso, algumas funções da rede tal como a própria engenharia de tráfego fica limitada devido às configurações dos *hardware* tão específicos. Logo, o *data plane* programável pode suprir essa falta: primeiro, por fornecer interoperabilidade do *data plane*, isto é, flexibilidade da engenharia de tráfego, segundo, por suportar novos protocolos e estruturas seguindo padrões abertos de desenvolvimento (MASOUDI; GHAFFARI, 2016).

Por centralizar o controle da rede numa entidade externa ao sistema, o operador de rede ou o desenvolvedor de sistemas, poderá, na camada de controle, fazer com que o sistema em tempo real tenha uma reação de acordo com um evento disparado através de gatilhos, ou até mesmo o *deploy* de novas aplicações ou serviços dinamicamente e também estaticamente por meio de troca de mensagens entre a unidade central e o *data plane*. SDNs são capazes de gerenciar toda uma rede através de uma visualização global com alta flexibilidade, entre eles: 1) alocação de recursos sob demanda; 2) serviços de computação na nuvem; 3) virtualização de rede, entre outros. Mais especificamente, capacidade de ampla abstração de topologias de rede, fornecendo um caminho fácil para configuração de dispositivos de rede, escondendo a complexidade dos *switches* convencionais com seus sistemas embarcados proprietários (LI; MENG; KWOK, 2016).

2.1.1 Componentes da Operação Básica

O *software* que implementa funções de rede tal como um equipamento de rede tradicional deverá cumprir com requisitos obrigatórios para o controle confiável do sistema. Esses requisitos podem ser catalogados independentemente da camada de interoperabilidade do conjunto pela troca de informações utilizando sistemas proprietários ou interface de programação de aplicativos aberta, do inglês, *Application Program Interface* (API). Esses componentes são fortemente acoplados e se comunicam através de *software* proprietário como mostra a Figura 2.O catalogo desse conjunto de APIs para compor um *data plane* são os seguintes:

- **Plano de Controle *Control Plane***: determina o caminho do fluxo de dados através dos *swtiches* autorizando ou desautorizando um pacotes, de acordo um conjunto de regras de saída estabecidos.
- **Plano de Encaminhamento *Forwarding Plane***: define o tratamento aplicado quando um dado entra no equipamento virtual de acordo com as decisões definidas no controle. Assim, executa funções de encaminhamento, enfileiramento e processamento baseado em instruções vindas do plano de controle.
- **Plano de Gerenciamento *Management Plane***: enquanto que o controle e o encaminhamento tratam acerca de definições de caminhos e regras para tráfego de dados, o gerenciamento trata acerca das falhas do sistema, configurando e monitorando ações sobre estes.
- **Plano de Operação *Operation Plane***: possui visão geral sobre o sistema permitindo um canal de comunicação de direto com o plano de gerenciamento disponibilizando informações para o gerenciamento. Além de enviar mensagens de *status* para o gerenciamento, permite atualizações através desse.

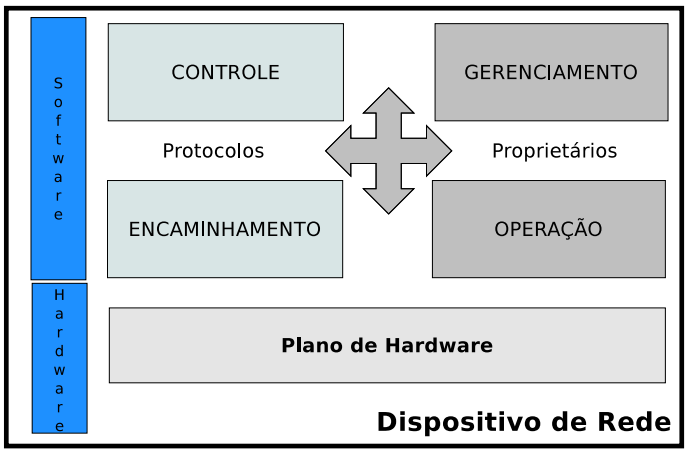


Figura 2 – *Data Plane* de Arquiteturas Convencionais

2.1.2 Arquitetura dos Sistemas SDN

SDN suporta programabilidade por meio de linguagens de programação de alto nível, onde, formas de troca de mensagens são utilizados por programas e aplicações para se comunicar com o sistema operacional da rede, ou algum outro programa de controle, seja uma ferramenta interna do controlador, ou um outro protocolo de comunicação, por exemplo HTTP REST. As chamadas de função dessas APIs podem requisitar sub-rotinas para sua execução utilizando padrões abertos ou padronizados. Para os padrões de programação que já são padronizados, por exemplo, OpenFlow, poderá garantir a interoperabilidade de controladores diferentes. Já para os padrões abertos, darão flexibilidade de adaptação para o contexto em que for aplicado

Controladores SDN pode ser programados diretamente em um servidor ou em servidores virtuais. O protocolo OpenFlow ou alguma outra API aberta poderá ser utilizada para obter o controle das *switches* no *data plane*. As entradas necessárias para as aplicações podem ser obtidas pelo controlador através do fluxo do tráfego dos equipamentos virtuais, essas informações de capacidade e demanda permitem que desenvolvedores e operadores de rede, possam implantar e escrever uma grande quantidade de aplicações para esses sistemas autônomos. Essa programabilidade pode ser alcançada através de duas interfaces, nomeadas como *northbound* e *southbound* interface.

2.1.2.1 Northbound Interface

Para prover a integração da camada superior da arquitetura SDN com a camada intermediária, temos uma API conhecida como *Northbound Interface*. Essa interface tem a função de conectar a camada de aplicação com sua camada subjacente, a camada de controle, permitindo que o controlador (*controller*) possua um canal de comunicação entre o controle e as aplicações de rede (SOOD et al., 2015). Uma aplicação de controle interno do sistema poderá reservar recursos através de múltiplos domínios de controle. Considerando o controlador como sistema operacional de rede, essa interface atuará como o caminho por onde as aplicações poderão interagir com o *hardware* (*switches*), coexistindo e interagindo com outras aplicações e utilizando serviços do sistema, sejam eles, Topologia, Descoberta da Rede, entre outros. Além disso, não exigirá que o desenvolvedor, ou o administrador, conheça a implementação de detalhes do controlador (AZODOLMOLKY, 2013).

A interface *Northbound*, possibilita a integração com plataformas de *Cloud Computing* do tipo *OpenStack* (LI et al., 2016) para uso de aplicações em rede, como *firewalls*, balanceadores de carga, sistemas de detecção de intrusão, *backups*, entre outros, possuindo uma grande variedade de APIs *northbound*. De acordo com Central (2016), a padronização dessa API ainda está em andamento, e o seu desenvolvimento é uma das metas a serem alcançadas pela *Open Networking Foundation* (ONF).

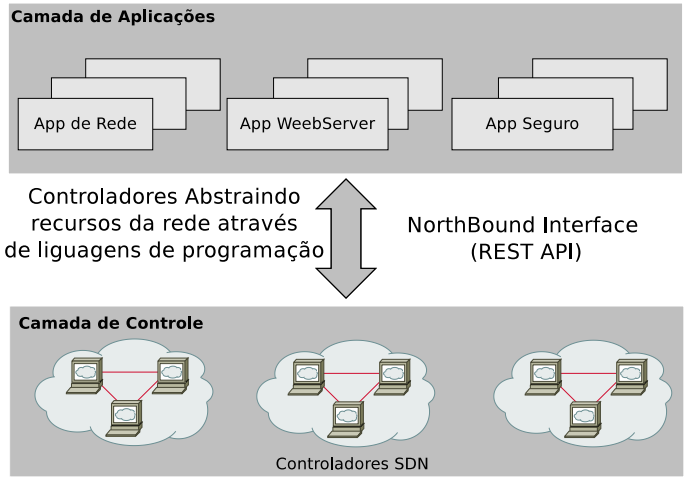


Figura 3 – Utilização de API para invocar o protocolo OpenFlow

A Figura 3, apresenta interação proporcionada pela interface *northbound* permitindo que o controlador possa executar requisições vindas da camada de aplicação mapeando como um serviço que aceita requisições. Dadas essas requisições, o controlador executa diretivas e comandos específicos para as *switches* e ainda fornece aplicações para o *data plane* de acordo com a topologia e as informações contidas nesses. Conforme Kreutz et al. (2015), um *control plane* deve suportar as seguintes funções especiais apresentadas na Figura 4:

- **Encaminhamento pelo Caminho mais Curto**: Utiliza informações das *switches* para definir rotas preferenciais.
- **Gerenciador de Notificação**: Recebe, processa e envia informações de entrada para aplicações, tal como: alarmes de notificação e segurança, e mudanças de estado.
- **Mecanismos de Segurança**: Fornece protocolos seguros entre aplicações e serviços.
- **Gerenciador de Topologia**: Realiza o *deploy* de topologias, construindo e mantendo conexões,
- **Gerenciador de Estatísticas**: Coleta Dados de tráfego da Rede.
- **Gerenciador de Dispositivos**: Configura parâmetros de *links* e gerencia a tabela de fluxos dos mesmos.

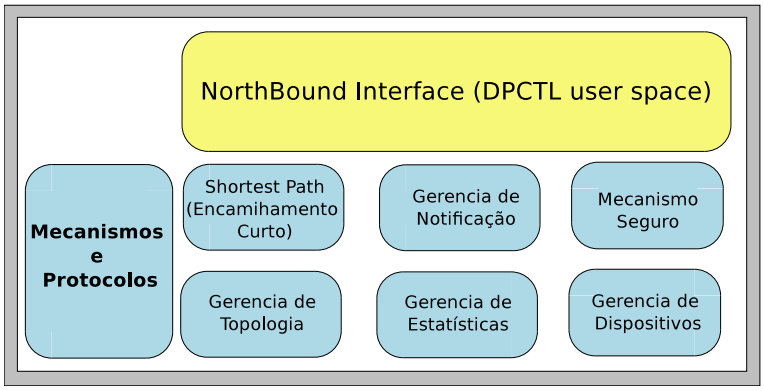


Figura 4 – Mecanismos e Protocolos suportados pela Interface NorthBound. Adaptado de (STALLINGS, 2015)

2.1.2.2 Southbound Interface

A interface *southbound* é a API que conecta logicamente a camada intermediária com os *switches* do plano de dados virtual, localizado na camada de infraestrutura. Além da conexão com os ativos de rede, essa API tem importância fundamental por remover claramente o controle dos concentradores, que em sistemas convencionais está embarcado nos equipamentos e são extremamente verticalizados. Existem modelos próprios e abertos de *southbound* API, tornando-a o maior desafio para a adoção do paradigma de redes programáveis, sendo o OpenFlow, o padrão aberto mais utilizado até o momento. Alguns modelos suportam apenas um único protocolo de comunicação, já em abordagens mais flexíveis, utilizam uma abstração dessa camada para poder disponibilizar para o *control plane* a troca de informações com o *data plane* enquanto suporta a utilização de múltiplas interfaces *southbound*.

Virtual Extensible LAN (VxLAN) é um desses padrões, que fazem mapeamento de dispositivos finais, sejam eles servidores ou máquinas virtuais, definindo quais estações farão parte do domínio de rede. Também permite o tunelamento entre controladores remotos através de interfaces físicas de *hypervisores* em VMs, e utilização de IPs flutuantes para conexões mestre/escravo, ou seja, se o tráfego de um *switch* virtual estiver muito alto, o protocolo poderá definir uma rota alternativa, abstraindo a visualização obtida da rede pelo controlador fixando um único domínio entre controladores conectados remotamente. Esse tipo de túnel também é possível utilizando o encapsulamento do roteamento genérico, do inglês, *Generic Routing Encapsulation* (GRE).

A API *southbound* OpenFlow prevê interoperabilidade, permitindo o *deploy* de aplicações SDN para diversos fabricantes, motivando o desenvolvimento de APIs *southbound* proprietárias por grandes fabricantes de *hardware*, como DELL, HP, Cisco, Juniper, Big Switch Networks, entre outros.

Existem outros projetos em torno de alternativas a *southbound interface* que são proprietárias. Essas alternativas são variações de protocolos já consagrados, tais como: SNMP, SSH, ou então o NETCONF para algo como a ferramenta de gerenciamento de *Switches* Open vSwitch Database Management (OVSDB) e OpenFlow Management and Configuration Protocol

(OF-Config) e também interfaces *southbound* com roteamento de borda através de protocolo BGP (EDELMAN, 2013).

A Figura 5 mostra a interação das APIs com a arquitetura SDN. Onde as requisições de usuários e o gerenciamento e controle são enviados para o controlador (orquestração) por meio da interface *northbound*, que recebe essas entradas e envia ações para os *switches* através da interface *southbound*. Por meio dessa interface, o controlador pode obter o status da rede e executar ações de gerenciamento sobre todo o domínio de uma forma centralizada.

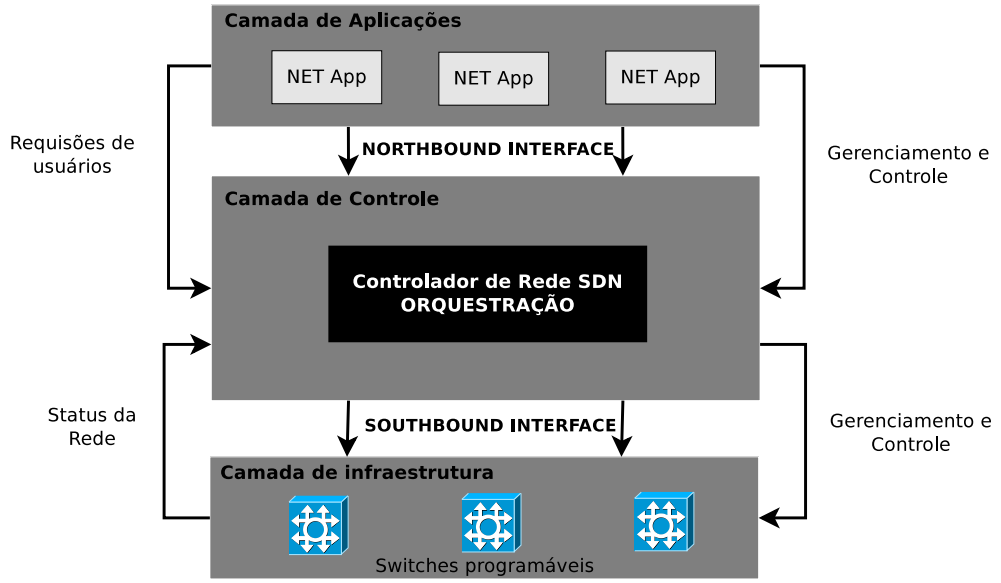


Figura 5 – Aplicação das APIs na arquitetura SDN. Adaptado de (ONF, 2016)

2.1.3 OpenFlow Southbound Interface

O protocolo Openflow é a API *southbound* aberta mais utilizada para implementação de *data planes* programáveis. Esse protocolo permite que sejam adicionados, consultados, modificados e removidos funcionalidades em *switches* e roteadores por meio de troca de mensagens e implantação de fluxos para execução de rotinas. Como já dito na Seção 2.1, a inteligência da rede é logicamente centralizada nos controladores baseados em *software* (no plano de controle), e os dispositivos da rede tornam-se simples encaminhadores de pacotes (no plano de dados), podendo ser programados por meio da API *southbound*. Um exemplo básico dessa funcionalidade, é a implantação de um fluxo para determinar uma porta de entrada (origem) e uma porta de saída (destino) mapeando o caminho de acordo com requisitos pré-estabelecidos, ou implantando fluxos para comunicação de uma porta com a outra dado um determinado evento.

Esse conjunto de protocolos e a API são aprovados de acordo com a RFC7426 do Internet Engineering Task Force (IETF). Cada *switch* mantém uma ou mais tabelas de fluxo, que são utilizadas para determinar como a comunicação será realizada. Assim, o controlador através dessas interfaces compatíveis com o protocolo OpenFlow, faz a comunicação com os *switches* para gerenciá-los e determinar padrões de comportamento. Dessa maneira, *switches* e roteadores

passam a ser controlados pelo dispositivo central desacoplando o *control plane* do *data plane*.

Segundo Nadeau e Gray (2013) o protocolo OpenFlow está dividido em duas partes como mostra a Figura 6:

- Um protocolo de comunicação (atualmente na versão 1.5.0) para estabelecimento de uma sessão de controle, definindo a estrutura da mensagem para modificações na troca de fluxos (*flowmods*) e coleta de estatísticas, e definição de uma estrutura fundamental para funcionamento de portas e *switches*.
- Um protocolo de gerenciamento e um configurador, para alocar fisicamente as portas do *switch* no controlador, definindo comportamentos ativo/inativo, e mensagens de falha de comunicação entre *switch* e controlador.

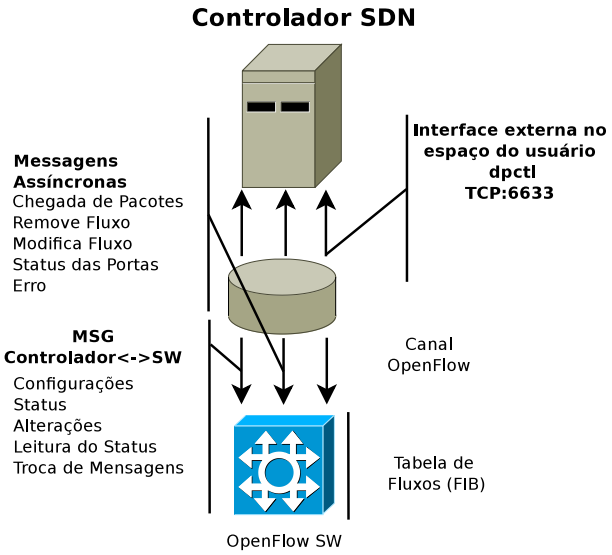


Figura 6 – Canal de comunicação utilizando o protocolo OpenFlow

Assim, os componentes chave do modelo OpenFlow, como mostra na Figura 6, tornaram-se parte integrante das definições de SDN, principalmente por:

- Separação do controle e do plano de dados;
- Uso de protocolos padronizados entre o controlador e um agente como elemento de rede instanciando os estados (no OpenFlow o *forwarding state*);
- Provimento da programabilidade da rede de um ponto centralizado através de uma API extensível.

2.1.3.1 Composição dos Componentes do OpenFlow

A Figura 7 enumera a composição do protocolo *openflow* e a funcionalidade de cada parte dessa interface. Os principais componentes que são utilizados para estruturar o *data plane* são os seguintes:

1. ***Openflow Controller***: compõe a inteligência do protocolo, realizando as decisões que serão encaminhadas aos *openflow switches*. Essas decisões ficarão armazenadas na tabela de fluxo dos *switches* no formato de ações. Tipicamente, o protocolo controlador executa adições, deleções e modificações por cada fluxo existente no *openflow switch*. Assim, o controlador é consultado, e no caso haja tráfego de pacotes desconhecidos, ações de deleção (*drop*) podem ser encaminhadas pelos controladores, ou até mesmo tempo de permanência do fluxo no *switch openflow*.

2. ***OpenFlow Switch***: dispositivo que funciona da mesma forma que um concentrador de redes convencional. Sua estrutura é utilizada somente para o encaminhamento de pacotes, sendo desprovido de qualquer sistema. O controlador se encarregará de controlar o caminho (*path*) e as ações para os pacotes de origem e destino dentro do *data plane*.

3. ***OpenFlow Protocol***: funciona na forma cliente/servidor, onde através de um canal seguro estabelece o enlace com o controlador e os *switches*. Pode ser análogo a qualquer protocolo de rede, pois trata-se de uma fusão de vários protocolos.

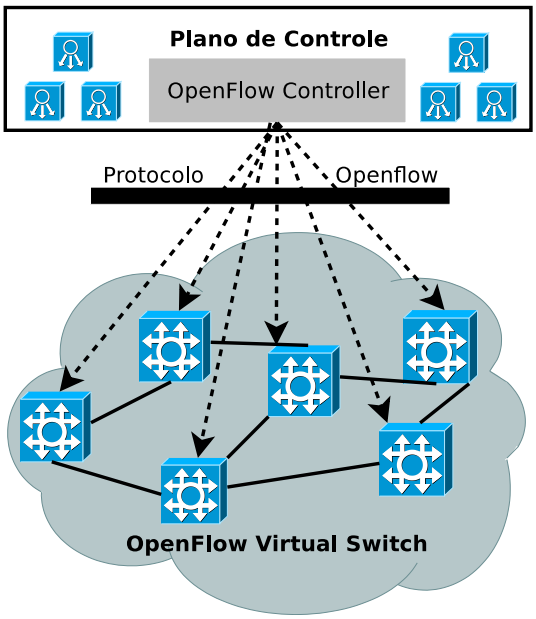


Figura 7 – Composição de Componentes OpenFlow

2.2 Information-Centric Networking (ICN)

As Rede de Computadores e a Internet foram projetadas para comunicação entre pares. Com o crescimento exponencial de acessos a conteúdos disponibilizados pela rede, uma grande quantidade de dados é trocada continuamente entre clientes e servidores. No entanto, devido à própria arquitetura do sistema, os mesmos dados são passados de *host* para *host* utilizando a mesma banda de rede repetidamente, causando sobrecarga de recursos de rede desnecessariamente. Além da elevação do consumo da banda, há também uma diminuição da Qualidade do

Serviço (QoS) da rede. Logo, para controlar esse conteúdo que passa redundante pelo enlace, o Information-Centric Network (ICN) vem sendo proposto (AHLGREN et al., 2012) (JACOBSON et al., 2009).

No paradigma ICN, o conteúdo é trocado pela rede sem o conhecimento de sua localização (*content centric network*), a partir dessa troca, o conteúdo é replicado entre os nós da rede sendo disponibilizado em cache para acessos posteriores. Dessa maneira, caso a mesma requisição seja solicitada ao sistema, o conteúdo será entregue pelo nodo da rede que estiver mais próximo do requisitante (SHAILENDRA et al., 2015). Com isso, um menor custo do roteamento entre a origem e destino a fonte do conteúdo será estabelecido, diminuindo o *overhead* de todo o sistema autônomo.

De acordo com Mougy (2015), ICNs são baseadas em três principais contextos: encaminhamento de conteúdo, *caching* intra-rede e envio *multicast*. Para implementação desses contextos, o *Software Defined Networking* (SDN) vem sendo proposto como a mais adequada estrutura para este fim por conta de sua capacidade de implementação, e da centralização de toda a inteligência da rede em um único dispositivo reprogramável e centralizado.

Entretanto, nesse sentido a integração dos dois paradigmas (SDN e ICN) se torna um dos grandes desafios para a comunidade científica das áreas de redes de computadores e sistemas distribuídos, onde além da flexibilização dos acessos e do encapsulamento do sistema ponto a ponto, também agregar as tecnologias de *cloud computing* para abranger a virtualização, utilizando ferramentas da nuvem, tanto para suportar o armazenamento de conteúdo massivo que trafega nos segmentos de rede, como para interligar complexos sistemas autônomos, tornando os *data planes* transparentes e com redundância de conteúdo provido pela computação na nuvem.

A Figura 8 apresenta um modelo de comunicação de clientes que fazem requisição a sistemas de conteúdo. Os clientes fazem requisições através de canais de comunicações não confiáveis (Internet) e obtém conteúdo armazenado em caches replicados pelos nodos da rede (*on-path*) conforme Seção 2.1.3.1.

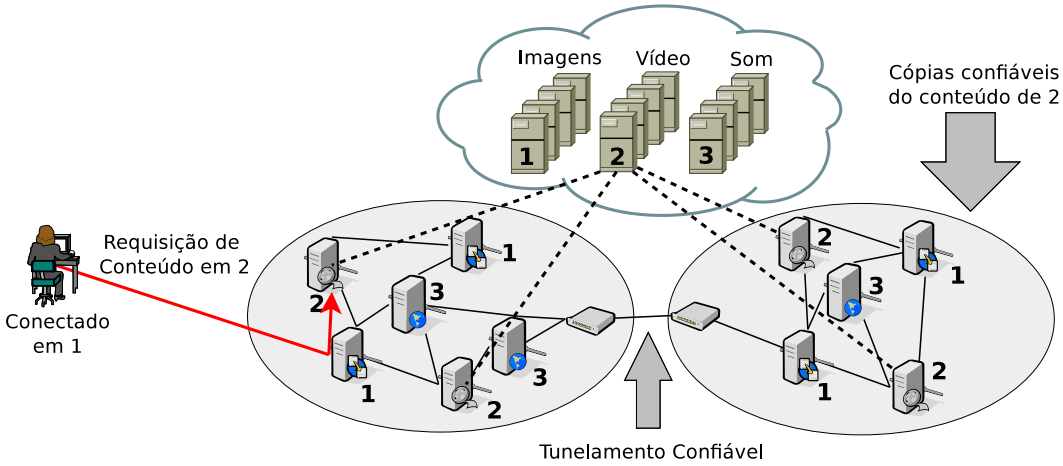


Figura 8 – Modelo de Comunicação ICN:Lado Cliente. Adaptado de (AHLGREN et al., 2012)

2.2.1 Características de Sistemas ICN

As iniciativas existentes estão direcionando seus esforços no desenvolvimento de uma arquitetura de Internet que pretende substituir o modelo atual, entregando soluções para problemas que são identificados constantemente na infraestrutura atual, tal como limitação de banda, encaminhamento comprometido, riscos de segurança, etc. O fato é que qualquer pacote que circule em um determinado segmento de rede, este deverá ser direcionado para algum destino, motivando problemas que podem levar ao consumo de recursos do sistemas desordenadamente, por exemplo, suporte para distribuição de conteúdo escalável, mobilidade, segurança, e canal de transmissão integro (XYLOMENOS et al., 2014). Para compor esse conjunto de melhorias, e embora ainda estejam em desenvolvimento ativo, essas arquiteturas ICN abordam um conjunto de funcionalidades principais com diferentes abordagens. Abaixo, identificamos os projetos principais, que constituirão a base tecnológica para que uma rede ICN possa ser suportada por SDN.

- Data Oriented Network Architecture (DONA) (KOPONEN et al., 2007).
- Content Centric Network (CCN) (SUN; ZHANG; ZHANG, 2016).
- Publish Subscribe Internet Routing Paradigm (PSIRP) (TROSSEN; PARISIS, 2012). Atualmente o projeto recebeu o nome de Publish Subscribe Internet Technology (PURSUIT)(FP7, 2017).
- Network of Information (NetInf) do projeto para o futuro da Internet (4WARD) (AHLGREN et al., 2010). Atualmente o projeto está intitulado de Scalable and Adaptive Internet Solutions (SAIL).

Os projetos para ICN diferem em relação aos seus detalhes, eles compartilham muitos pressupostos, objetivos e propriedades arquitetônicas. O objetivo é desenvolver uma arquitetura de rede que seja mais adequada para acessar e distribuir eficientemente o conteúdo, sobre o uso que prevalece atualmente das redes de comunicação, e que poderá melhor lidar com desconexões, interrupções e efeitos de inundação de conteúdo no mesmo enlace (AHLGREN et al., 2012).

Segundo estudo de Xylomenos et al. (2014), sistemas ICN devem possuir quatro funcionalidades principais. Essas funcionalidades são descritas logo abaixo:

- **Nomeação**: O acesso ao conteúdo deverá ser feito por nomes, isto é, nomes de conteúdo são requisitados e são acessados independentemente de sua localização geográfica no *data plane*. Dependendo da arquitetura utilizada seja DONA, CCN, PSIRP ou SAIL o nomeamento poderá ou não ser humanizado.
- **Resolução de Nomes e Roteamento dos Dados**: A resolução de nomes envolve a associação de um nome de conteúdo com seu arquivo físico armazenado em algum lugar

do sistema. Enquanto o roteamento deverá estabelecer o caminho para que esse arquivo chegue até o solicitante. Em ICNs, poderão ser usadas duas abordagens: Acoplado e Desacoplado. Na abordagem acoplada a requisição será encaminhada para um servidor que devolverá o conteúdo pelo mesmo caminho de origem. Na abordagem desacoplada a resolução de nomes poderá entregar para o roteamento, e este poderá determinar qual a melhor rota a seguir, não necessariamente pelo mesmo caminho de origem.

- **Cache**: Duas abordagens são incentivadas, *On-Path* e *Off-Path*. No primeiro modo, o cache será distribuído pelos nodos da rede. Enquanto no último, o cache deverá ser entregue por uma unidade central que fará o armazenamento do cache do sistema. Alguns sistemas de proxy reverso utilizam essa abordagem.

- **Mobilidade**: A mobilidade é essencial em arquiteturas ICN, uma vez que a cada nova requisição o conteúdo deverá ser disponibilizado de forma transparente para o requisitante, escondendo a complexidade. Já para mobilidade de caches, esse suporte é um desafio, visto que a resolução de nomes (*on-path*) e as tabelas de roteamento (*off-path*) precisam ser atualizados.

- **Segurança**: Agentes seguros que possam estabelecer relação de confiança com o sistema de resolução de nomes e os controladores de roteamento para verificar se o conteúdo que retorna para os requisitantes, são realmente o que foi requisitado.

2.2.2 Arquitetura ICN

Quase todas as arquiteturas ICN possuem o modelo de recuperação de dados baseados nos problemas de nomeação de conteúdo da Internet, bem como a falha de serviços de rede devido ao *overhead* do enlace. Segundo Akbar et al. (2014) as arquiteturas ICN podem ser categorizadas em três categorias, como apresentadas na Figura 9.

- **Horizontais (*Flat-Based*)**: Independe de localização e utilização de *string* única para localização do conteúdo. Arquiteturas orientadas a dados (DONA), que utilizam pares de identificadores (PSIRP e PURSUIT) são classificados como horizontais, onde além da disseminação do identificador pela rede, o cache precisa ser armazenado em cada rota por onde passa.

- **Hierárquica (*Hierarchical*)**: O conteúdo será assinado tal como as URLs da Internet, contribuindo para o roteamento, uma vez que um nodo já possuí o conteúdo requisitado não há necessidade de um novo encaminhamento, otimizando os recursos de conexões e de rede.

- **Híbrida (*Hybrid-Bases*)**: Permite que servidores de conteúdo disponibilizem dados para os requisitantes. Por exemplo, um conteúdo ser assinado unicamente para o requisitante.

Nessa categoria a arquitetura pode ser horizontal ou hierárquica e o objeto do dado pode ser identificado por identificadores IDs e por seu nome. Arquiteturas SAIL são baseadas nesse modelo.

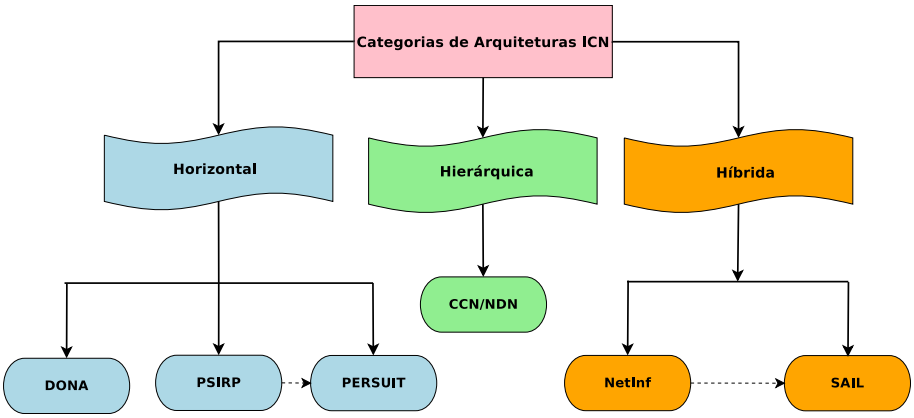


Figura 9 – Taxonomia da arquitetura ICN. Adaptado de (AKBAR et al., 2014)

2.2.2.1 Rede de Dados Nomeada (*Named Data Network (NDN)*)

Estabelece rotas para busca de conteúdo (*Content Router (CR)*) onde recebe pedidos de usuários em busca de conteúdo. Esta arquitetura não tem noção da localização do respondente do pedido no seu nível mais baixo (JACOBSON et al., 2009). Ao contrário das redes IP, o ICN não precisa do Uniform Resource Locator (URL) para localizar a informação, nem da estrutura do arquivo para nomear o arquivo. No entanto, preserva-se o projeto das redes convencionais que tornam o TCP/IP simples, robusto e escalável.

2.2.2.2 Arquitetura de Rede Orientada a Dados (*Data Oriented Networking Architecture (DONA)*)

Esta arquitetura tem a capacidade de armazenar o cache na camada de rede trazendo vantagens sobre as outras arquiteturas podendo ser utilizados por vários Sistemas Autônomos (SA) para propagar o cache através da rede, usando o conceito de "deixe uma cópia aqui"Leave Copy Down (LCD) de acordo com Abdullahi, Arif e Hassan (2015). No entanto, esta arquitetura usa endereçamento e roteamento TCP/IP, que introduzem na rede um controlador do cache e requisições, chamado Request Handler (RH). RH e roteadores estão incluídos no SA, operando de acordo com as políticas de roteamento aplicadas na rede. Assim, as funções de operação do DONA são semelhantes ao CCN com as principais diferenças de operação no envio de uma mensagem para registrar o objeto no RH do manipulador de solicitação, de acordo com Xylomenos et al. (2014).

2.2.2.3 Roteamento de Internet em Publicação Assianada (*Publish Subscribe Internet Routing (PSIRP)*)

Nesta arquitetura, os dados nomeados (*Named Data Objects (NDOs)*) também são publicados na rede pelas fontes do (NDO). Os objetos de informação são recuperados pelos assinantes depois que os objetos de informação são totalmente publicados na rede pelo editor (publicador). Nessa arquitetura, três novos nodos são introduzidos em cada AS: o Rendezvous Node (RN) recebe e armazena publicações de dados e mensagens de interesse, o Gerenciador de Topologia (TM) estabelece caminhos de roteamento entre editores e assinantes, enquanto o Nó de encaminhamento (FN) executa reencaminhamento e armazenamento em cache.

2.2.2.4 Solução de Internet Adaptável e Escalável (*Scalable and Adaptive Internet Solutions (SAIL)*)

Essa arquitetura pode recuperar conteúdo de duas maneiras: a primeira, por resolução de nomes e a segunda por roteamento baseado em nome, que permitem a adaptação em diferentes ambientes de rede. Dependendo do modelo usado, as fontes publicam NDOs registrando uma ligação de nome/localizador com um serviço de resolução de nomes (*Name Resolution Service (NRS)*) ou anunciando informações de roteamento em um protocolo de roteamento. Em SAIL, um nodo que contém uma cópia de um NDO (incluindo caches locais) pode, opcionalmente, registrar sua cópia com um NRS, adicionando assim uma nova ligação de nome/ localizador. Alternativamente, o receptor pode enviar diretamente um requisição GET com o nome NDO, que será encaminhado para uma cópia NDO disponível usando o roteamento baseado em nome.

A Figura 10 apresenta um resumo visual de todas as arquiteturas nas seções anteriores, mostrando o fluxo de troca de dados e a requisição de conteúdo.

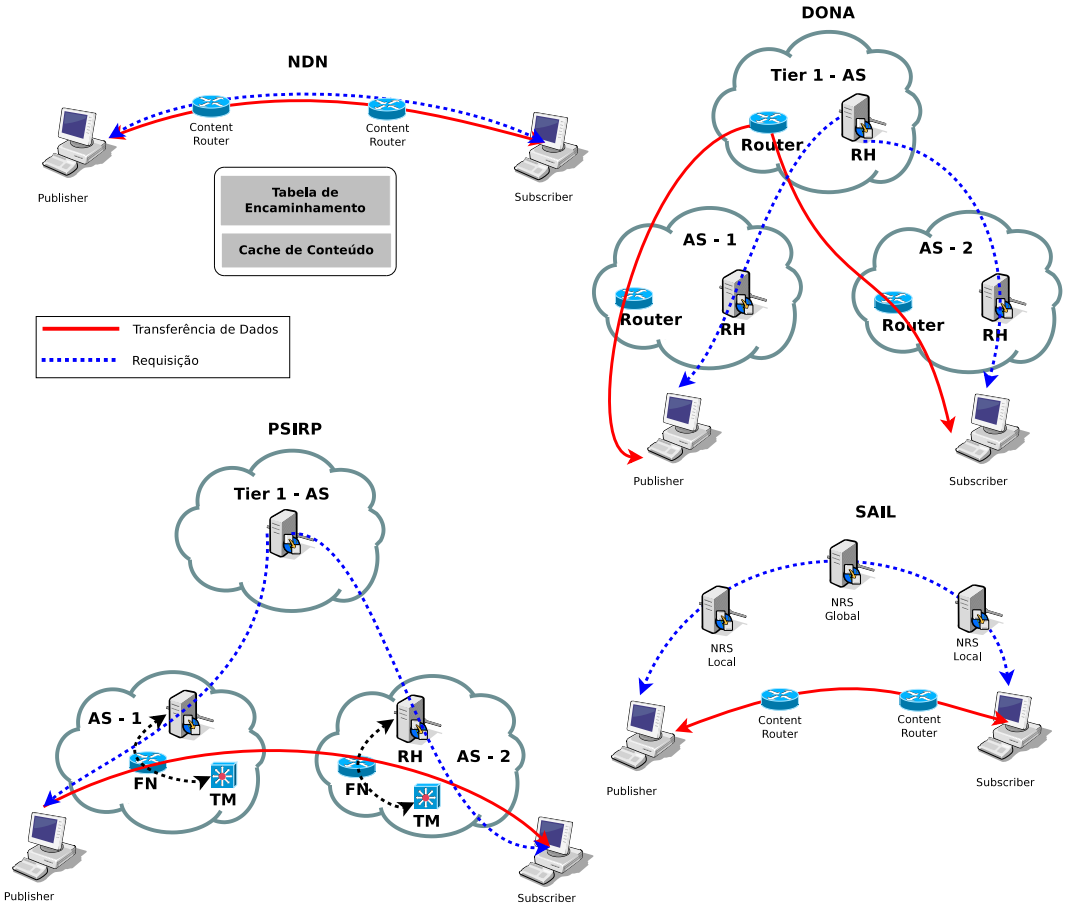


Figura 10 – Arquiteturas ICN (MOUGY, 2015)

.

2.3 Computação na Nuvem

A computação na nuvem pode ser definida como o uso de tecnologia computacional que aproveita o poder de processamento de inúmeros computadores interconectados ocultando a complexidade de redes e sistemas que operam em sua infraestrutura. A rápida disseminação do termo denominado "nuvem"se deve ao fato da computação estar presente na vida cotidiana de todos, seja local, móvel ou empresarial. A quantidade de *datacenters* que fornecem serviços de computação em nuvem ficou comprovado como uso da computação de utilidade, serviços seguros de armazenamento, computação independente de infraestrutura, computação escalável, etc. O acesso a esse tipo de recurso cresceu exponencialmente (ANAN et al., 2016) nos últimos anos.

Ainda de acordo com Anan et al. (2016), privacidade, integridade e segurança são aspectos cruciais quanto a utilização de serviços na nuvem, portanto é muito importante que os sistemas computacionais em nuvem garantam requisitos mínimos. A utilização de SDN vem amadurecendo rapidamente por conta da utilização de *datacenters* em nuvem segundo a ONF. Objetivando otimização da eficiência energética, SDNs podem analisar as condições de tráfego dos centros de dados e ajustar dinamicamente o conjunto de elementos de rede (HELLER et al., 2010).

Logo, para promover a adoção de sistemas em nuvem que possibilitem a criação de *datacenters* virtualizados que suportem ambientes experimentais, sistemas de nuvens privadas estão sendo disponibilizados utilizados, como: CloudStack (KUMAR et al., 2014), OpenNebula (MILOJIČIĆ; LLORENTE; MONTERO, 2011), OpenStack (OPENSTACK.ORG, 2016), entre outros. Essas infraestruturas como serviço, do inglês *Infrastructure as a Service (IaaS)*, controlam um grande *pool* de computação em nuvem, armazenamento e controlando os recursos de rede. Através de painéis administradores de rede e sistemas, podem disponibilizar recursos e operar a orquestração de uma nuvem privada quase que completamente, expandindo possibilidades de integração de controladores OpenFlow através de Servidores Virtuais Privados, do inglês, *Virtual Private Server (VPS)* (WANG et al., 2012).

Em nuvens OpenStack, por exemplo, na camada de rede (*network*) encontram-se os agentes de serviços para a rede da nuvem (RADEZ, 2015) que poderão disponibilizar recursos adicionais e um grande número de ferramentas. Isso inclui: escalabilidade de recursos da nuvem sob demanda; bancos de dados; mensageiro; e processamento *big data*. Além disso, cada um desses grupos poderá ser instalando separadamente em vários servidores, mas também poderá ser instalados em um ou dois servidores. Já para um ambiente em nuvem que possua VPS, através de uma grande de computação, um servidor independente pode ser instanciado com recursos disponíveis quase completamente, virtualizadores de rede e de controladores SDN podem instalados (*deployment*) e possibilitar comunicação de controladores remotos através de tunelamento com encapsulamento de roteamento genérico ou LANs extensíveis virtualmente.

A Figura 11 apresenta o contexto de um serviço de nuvem. Uma empresa mantém acesso de sua LAN interna ou um conjunto de LAN interligadas através de roteadores de borda. O serviço de computação em nuvem, mantém uma coleção de servidores que são gerenciados por meio de mecanismos de acesso remoto e gerenciamento, redundância, e ferramentas de segurança. Ainda na Figura 11 a nuvem é mostrada conectando através de roteador de borda um conjunto de *blade servers*.

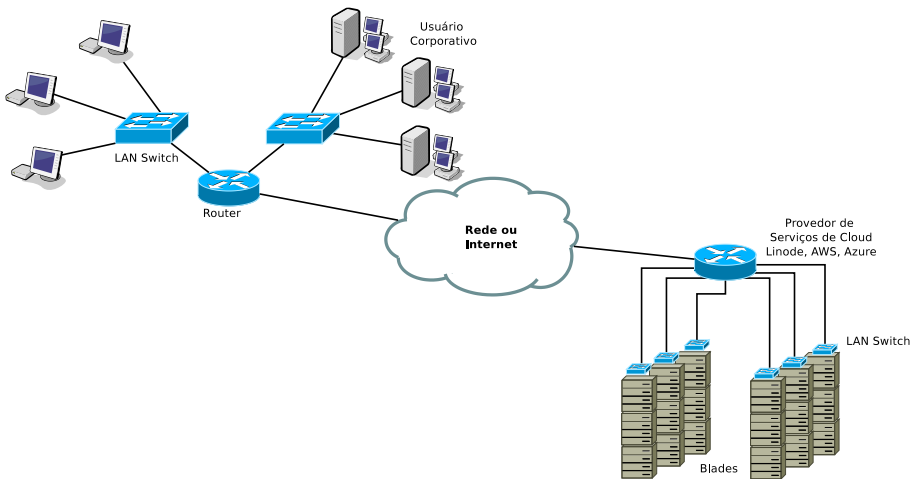


Figura 11 – Contexto Genérico de Serviço em Nuvem

2.3.1 Tipos de Nuvens

O modelo que habilita um grande *pool* de computação, redes computacionais sob demanda, acesso a recursos de computações configuráveis, tais como: redes, servidores, armazenamento, aplicações e serviços; podendo ser rapidamente implementados e disponibilizados com o mínimo de gerenciamento e interação é apresentado nesta seção. O modelo em nuvem é composto de cinco características essenciais, três modelos de serviço, e quatro tipos de implementação, segundo o National Institute Standard and Technology (NIST), SP-800-145. A Figura 12 apresenta o modelo sugerido pelo NIST para implementação da nuvens computacionais.

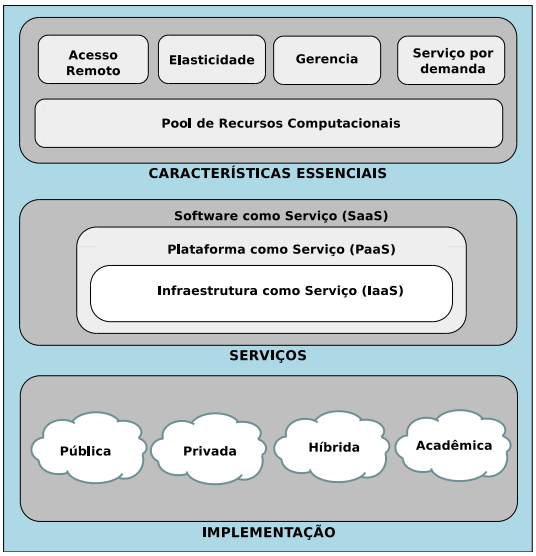


Figura 12 – Elementos da Nuvem, adaptado de (STALLINGS, 2015)

2.3.1.1 Nuvens Públicas (*Public*)

Um infraestrutura de nuvem pública, é desenvolvida para o contexto do público em geral ou um grande número de clientes corporativos. O mantenedor desse tipo de implementação, é responsável por todo o funcionamento da nuvem, seja contexto de infraestrutura ou contextos de dados, e gerenciamento de operações orquestradas pela nuvem e que funcionam internamente na nuvem. Em nuvens públicas o requisito mais importante é o sistema de *Firewall*. Nesse modelo, um único sistema se encarrega de controlar o tráfego de um sistema autônomo com um conjunto de sistemas que passam através do mesmo. Aplicações e armazenamentos podem ser disponibilizados através de endereços seguros via internet (IPSec), podendo ser gratuitos ou pagos dependendo da quantidade de transferência e de armazenamento desejado pelo consumidor do serviço. Este tipo de nuvem é de fácil utilização e suporta uma grande quantidade de serviços, por exemplo: aplicações de texto na web como: Amazon, Google e Office365; Serviços de armazenamento gratuito, como Google Drive, Outlook e Asus Web Storage; E sistemas de e-mail que destinam uma parte para o armazenamento de fotos.

2.3.1.2 Nuvens Privadas (*Private*)

A infraestrutura de nuvens privadas são implementadas internamente juntamente com o ambiente de T.I. organizacional. A organização terá autonomia para gerenciar seu espaço computacional, gerenciando servidores e *storages* de acordo políticas pré-estabelecidas. Nuvens Privadas podem entregar modelos de serviços Infrastructure as a Service (IaaS) através de internet ou intranet conetando a organização através de redes virtuais, como Virtual Private Network (VPN), como também aplicação e suítes completas de aplicativos. Por exemplo, Microsoft Office365 on-line, Gerenciamento de contas de e-mail Corporativo on-line, Armazenamento Ilimitado e Bancos de Dados sob demanda on-line.

2.3.1.3 Nuvens Acadêmicas (*Academic*)

Infraestruturas de nuvens acadêmicas, possuem características de nuvens privadas e públicas, mas não são consideradas híbridas. Com características privadas, esse tipo possui acesso restrito para algumas áreas, por exemplo, gerencia de níveis de acesso para usuários da nuvem. Com características públicas, os usuários tem acesso irrestrito dentro do perfil de utilização da conta. Por exemplo, de modo privado: criação de interfaces de rede virtuais; de modo público: configurações de contas de e-mail institucional para alunos.

2.3.1.4 Nuvens Híbridas (*Hybrid*)

As infraestruturas de nuvens híbridas são compostas de pelo menos de dois ou mais modelos, sejam eles públicos, ou privados. A diferença é que essas nuvens são interligadas através de tunelamentos seguros possibilitando troca de dados aplicações, por exemplo, *load balance* entre diferentes nuvens. Nesse modelo, informações sensíveis podem ser armazenadas no lado da nuvem privada, enquanto informações em comum que podem ser acessadas por outras entidades que compartilham das mesma nuvem, poderão ser armazenadas no lado público. A Tabela 1 apresenta a representação das implementações descritas, comparando requisito não funcionais da nuvem.

Tabela 1 – Comparação dos Tipos de Implementação de Nuvens. Adaptado de (STALLINGS, 2015)

	Pública	**Privada**	**Acadêmica**	**Híbrida**
Escalabilidade	Muito Alta	Limitada	Limitada	Muito Alta
Segurança	Moderada	Muito Seguro (Opcional)	Muito Seguro	Muito Seguro
Performance	Baixa	Muito Alta	Muito Alta	Boa
Confiabilidade	Média	Muito Alta	Muito Alta	Muito Alta

2.3.2 Modelos de Serviços

2.3.2.1 Software como Serviço (SaaS)

SaaS abrange a capacidade de fornecimento de serviços de *software* para serem acessados especificamente em nuvem através de uma interface remota ou mesmo de um *browser*. Além do dos serviços de *software* (MELL; GRANCE et al., 2011), SaaS permite que usuários tenham acesso a VPSs comerciais sob demanda, para implementação de LANs internas ou mesmo interfaces rede virtuais para serem disponibilizados utilizando a infraestrutura de uma nuvem. Por exemplo, configurar Servidores Web virtuais para serem acessados remotamente, por meio de terminais seguros, tal como: SSH, entre outros.

Segundo o Mell, Grance et al. (2011), a disponibilização de uma aplicação para controlar uma VPS em nuvem, não eleva a permissão de um usuário com acesso completo. O usuário da instância somente tem permissão na grade de configuração contratada, logo, configurar sistemas operacionais da nuvem, gerenciar discos reais da nuvem, ou instalar uma outra placa de rede na nuvem não são possíveis. A configurações do usuário ficam restritos somente dentro da sua conta.

A Figura 13 apresenta um modelo de serviço SaaS baseado em SDN, onde tem-se um Datacenter SDN instalado sobre um serviço de VPS comercial, ou seja, uma instância de *software* contratado. A orquestração é feita de forma remota e centralizada, acessando a grade de configuração através de um *Dashboard* via web browser, ou acesso de administrador por interface SSH (acesso de usuário Root).

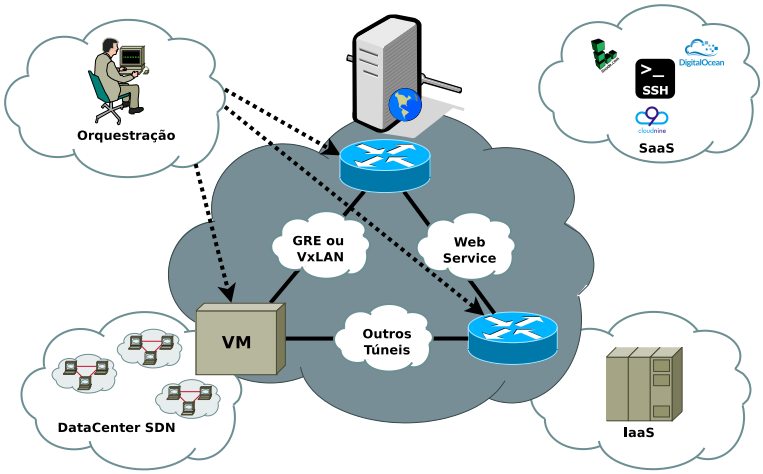


Figura 13 – Software Defined Wide Area Network (SDWAN)

2.3.2.2 Plataforma como Serviço (PaaS)

Uma infraestrutura PaaS, disponibiliza plataformas sob demanda para construção de aplicações que serão desenvolvidas pelos usuários deste serviço. Receitas de PaaS, poderão executar aplicações sob um vasto número de ferramentas de desenvolvimento, com suporte a linguagens de programação de alto nível, *Runtimes* de maquinas virtuais (ex.: JVM) e outros

mecanismos que irão possibilitar a implementação de novos aplicativos.

Dessa forma, é possível realizar o *deploy* de receitas de suítes de desenvolvimento completas sem a necessidade de configuração de *hardware* físico, fazendo com que um ambiente de desenvolvimento, seja "destruído"ou "remontado"sem grande esforço. A Figura 14 apresenta uma exemplo de arquitetura PaaS.

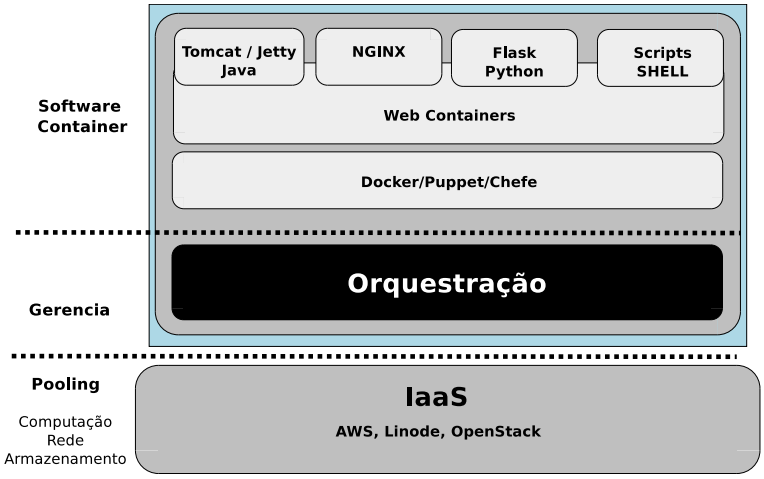


Figura 14 – Arquitetura PaaS

De acordo com Stallings (2015) uma lista dos principais serviços de uma nuvem PaaS é descrito como segue:

- **Big Data as a Service (BDaaS)**: Ferramenta para análise complexa de cruzamento de grandes massas de dados.
- **Business Intellingence (BI)**: ferramentas para criação de *dashboards*, relatórios e análises de quantidades massivas de dados.
- **DataBase as a Service (DBaaS)**: suportar uso de Sistemas Gerenciadores de Bando de Dados (SGBDs) e também soluções para banco de dados não-relacionais.
- **Desenvolvimento e Teste**: Ferramentas específicas para teste de ciclos de desenvolvimento.
- **Serviços de Integração**: Serviços para integrar aplicações de uma nuvem para outra nuvem.
- **Suíte de Desenvolvimento Integrado**: plataformas de desenvolvimento, disponibilizando um banco de dados e o *runtime* específico para cada ambiente. Por exemplo: XAMP.

2.3.2.3 Infraestrutura como Serviço (IaaS)

Na Infraestrutura como Serviço, traz possibilidades de provisionamento sob demanda para provisionar processamento, armazenamento, redes e outros sistemas específicos, de modo

que usuários da nuvem possam processar, armazenar, criar redes virtuais utilizando os recursos computacionais de VPSs, instalando *software* e outras aplicações (MELL; GRANCE et al., 2011). Uma vez contratado o *pool* IaaS, o próprio usuário vai alocar espaço em disco para sistemas de arquivos, bem como administrar seu IP Público, possibilitando a criação de LANs internas.

Tipicamente, usuários podem provisionar sua infraestrutura, utilizando painéis de gerenciamento web disponibilizado pelo próprio fornecedor de serviço IaaS. Como exemplo desse painel de configuração baseado no modelo IaaS são: Amazon Elastic Compute Cloud (Amazon EC2), Microsoft Azure, RackSpace, Linode, entre outros. A Figura 15 apresenta um exemplo genérico da arquitetura IaaS.

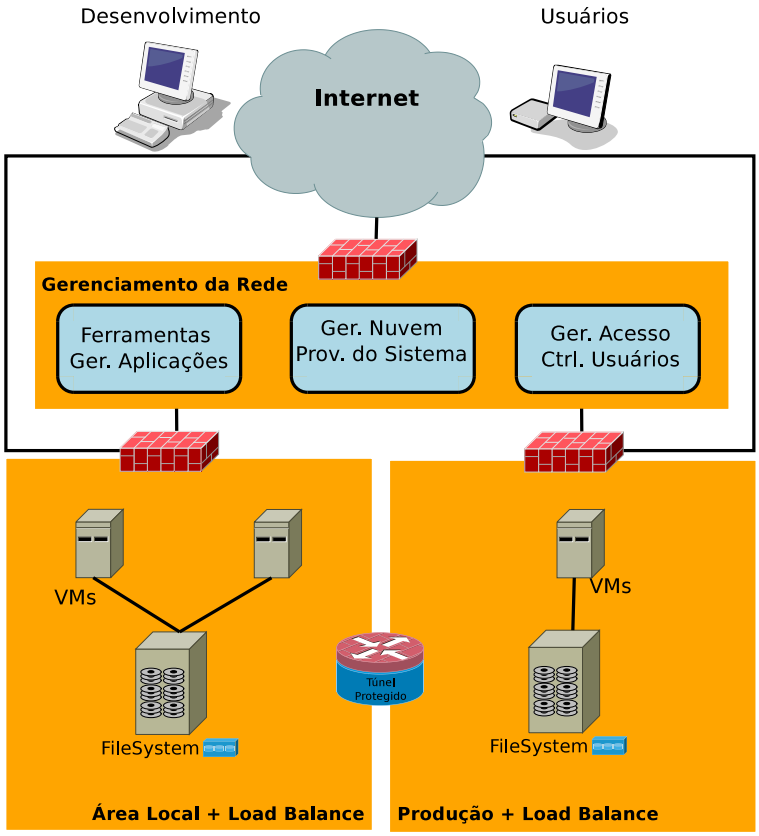


Figura 15 – Arquitetura IaaS

Ainda de acordo Stallings (2015), uma nuvem que possua característica de arquitetura IaaS poderá prover os seguintes serviços:

- **Recuperação e Backup**: suporte a *backup* de arquivos, *filesystem* e servidores.
- **Provisionamento de Computação**: suporta a elasticidade de recursos computacionais sob demanda.
- **Redes de Conteúdo**: disponibilizar recursos para implantação de rede de entrega de conteúdo, tal como *Information Centric Networks (ICNs)*.

- **Armazenamento *Storage***: suportar quantidades massivas de armazenamento com capacidade de serem utilizados para aplicações, *backup*, arquivamento, *storage*, entre outros.

2.3.3 Integração de SDNs em Nuvem

O sistema básico de gerenciamento de redes virtuais pode ser disponibilizado por um serviço de *cloud computing*. Tecnologias de redes de sobreposição por virtualização e SDN, podem ser utilizados para estabelecer um esquema de tunelamento confiável e eficiente para interligar redes SDN remotas através de controladores conectados. Diferentes funcionalidades do gerenciamento de redes virtuais podem ser suportados nos três tipos principais de *clouds* de acordo com a Seção 2.3.1.

Em nuvens privadas, a conta de administrador pode suportr o controle das redes que são internas à nuvem. Em nuvens públicas, a própria nuvem suporta o controle das redes internas à nuvem, mas com isolamento dos sistemas autônomos. Em nuvem híbridas (no caso dessa pesquisa), a nuvem suporta o controle interno e massivo, e ainda permite que o usuário, em seu espaço computacional, possa ter controle sob suas redes virtuais, criando os próprios túneis através da nuvem (ZHANG; LI, 2016).

2.3.3.1 Tunelamento em Nível de Enlace(L2) e Rede(L3)

Um túnel de camada de rede, é uma interface lógica em um roteador multicamada que suporta o encapsulamento de pacotes dentro de um protocolo de transporte. É uma arquitetura projetada para prover o esquema de encapsulamento ponto a ponto (*Point–to–Point*). Dessa forma é possível criar uma rede lógica entre duas Máquinas Virtuais (*Virtual Machines (VMs)*) em diferentes redes, ou seja, conectar duas ou mais redes de camada 3 (*Layer 3 (L3)*), fazendo com que essas redes estivessem operando em camada 2 (*Layer 2 (L2)*). Logo, cada uma faz parte de sua própria rede, mas pertencentes ao mesmo domínio. Através de um identificador de 24 bits, é possível criar 16,7 milhões de redes utilizando LANs Virtuais Extensíveis, do inglês *Virtual eXtensible Local Area Network* (VxLAN) ou, um número muito maior que o limite de 4096 das VLANs devido ao uso de um identificador de apenas 12 bits (ZHANG; LI, 2016).

2.3.3.2 Encapsulamento de Roteamento Genérico

Generic Encapsulation Routing (GRE) é um protocolo de camada 2, *Layer 2 (L2)* que encapsula protocolos da camada superior L3. Os túneis GRE assim como VxLAN, IP *Security*(IPSec), *Virtual Private Network*(VPN) são projetados para transferir dados através de IP público ou redes intermediárias. Os roteadores usam GRE para enviar tráfego através de uma rede interativa que não suporta esse tráfego.

Um exemplo da falta de suporte para alguns protocolos de rede é a Internet. A rede mundial de computadores não faz roteamento de tráfego *multicast*, isto é, não é possível trocar mensagens com um grupo específico de sistemas autônomos. Logo, a utilização de túneis genéri-

cos, possibilita múltiplas trocas de mensagens e atualizações por meio de GREs encapsulados na rede internet. O tunelamento genérico ou LANs virtuais extensíveis possibilitam tal condição. Ainda há possibilidade de utilização de túneis GRE para enviar tráfego no-IP pela Internet. Essas mensagens, são de grande importância para os estudos sobre a adaptação das redes ICN para o contexto da Internet. A Figura 16, apresenta um modelo básico de cabeçalho GRE que conecta duas redes distintas por intermédio de uma rede virtual, fazendo a junção entre esses sistemas.

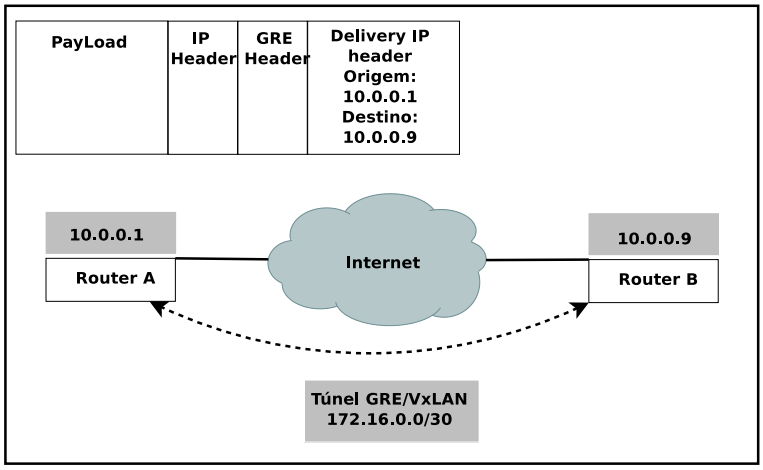


Figura 16 – Tunelamento GRE

2.4 Revisão Sistemática da Literatura

De acordo com Kitchenham (2004) a Revisão Sistemática da Literatura (RSL)(do inglês *Systematic Literature Review (SLR)*) tem o objetivo de fornecer estrutura para o desenvolvimento de pesquisa baseadas em evidências, a partir da seleção, sintetização e catalogação das pesquisas mais relevantes de acordo com um tema de pesquisa que possa dar contribuição ao pesquisa científica e tecnológica. Além disso, a SLR deve promover a identificação, avaliação e interpretação, de todas as pesquisas disponíveis em fontes de consulta relevantes para a investigação de área de interesse ou fenômeno, bem como comparar todos os resultados escolhidos para uma determinada questão de pesquisa.

2.4.1 Método de Planejamento da Revisão

Conforme dito na Seção 2.1, o tema SDN recebeu muita atenção nesses últimos anos da comunidade científica como também da indústria. Existem diversas propostas para desenvolvimento de arquiteturas SDN que suportem o uso das redes ICN, de padrões já utilizados na construção desses modelos, ou até mesmo, estudos complementares ao protocolo OpenFlow. Outros estudos, direcionam para a integração de redes SDN com redes convencionais, ambas as abordagens apontam para novas arquiteturas. Dessa forma, para embasar o projeto, uma revisão sistemática da literatura baseado no modelo de Kitchenham (2004) e Wazlawick (2014) faz-se

necessário, com o objetivo de identificar, analisar e avaliar, estudos que viabilizem a integração de SDN com ICN e posteriormente o estudo e desenvolvimento de um nova arquitetura de redes cêntricas que englobe a redundância de informações volumes de armazenamento configurados em nuvens privadas.

2.4.2 Método de Condução da Revisão Sistemática

O método proposto pelos autores em 2.4.1, servirá de base para os seguintes estágios executados na condução da revisão sistemática da literatura:

1. Identificação da pesquisa;
2. Seleção dos estudos primários;
3. Classificação dos estudos por relevância (alto, médio ou baixo);
4. Extração dos estudos necessários;
5. Sintetização dos estudos capturados.

Questões de Pesquisa

1. Existem mecanismos capazes de possibilitar a construção de uma Information Centric Network (ICN) utilizando infraestrutura de redes baseadas em *software*, por exemplo, *Software Defined Networking (SDN)*?
2. Quais são as ferramentas necessárias para o desenvolvimento de arquiteturas para suportar redes de conteúdo?
3. A otimização de requisições por meio de caches integrados com plataformas em nuvem se faz realmente necessário?

População

- Publicações acerca da integração entre SDN e ICN.

Intervenção

- Técnicas de desenvolvimento de redes de conteúdo flexíveis.

Resultados

- Iniciativas identificadas na pesquisa.

Definição de critérios e seleção das fontes

- Adoção dos seguintes critérios para seleção de estudos primários:
  - Disponibilidade de consulta dos artigos através de bases disponíveis na web;
  - Uso de indexador de mecanismos de busca com campo de busca e configuração padrão de título, *abstract* e palavras-chave;
  - Consultas nas bibliotecas acadêmicas virtuais.

String de Busca

- Foi utilizado mais de uma string de busca nesta revisão sistemática, por conta do retorno nulo de informações com a configuração da busca de informações que se pretendia atingir. As seguintes strings de busca foram utilizadas com apresentado na caixa de texto abaixo:

```
(TITLE-ABS-KEY ( software,defined,networking ) AND,
TITLE-ABS-KEY ( information,centric,network ) AND,
TITLE-ABS-KEY ( sdn ),AND,TITLE-ABS-KEY ( icn ) AND,
(TITLE-ABS-KEY ( software defined networking ) AND,
TITLE-ABS-KEY (information centric networks ))
```

Critérios de inclusão e exclusão

- As informações necessárias foram acessadas na web, e também foi considerado para a pesquisa a busca manual em livros sobre o assunto.
  - Artigos disponíveis nas principais bases de informação acadêmicas com foco na área da computação;
  - O material deve apresentar o texto completo sobre o assunto em formato de papel ou meio eletrônico;
  - Trabalhos escritos em inglês (preferencialmente) e português;
  - Os artigos devem apresentar a integração de redes programáveis com redes de conteúdo;
  - Todos os anos foram considerados na busca;

Execução do protocolo de RS

A estratégia utilizada combinou palavras e frases que podem ser encontradas no título, *abstract* e palavras-chave para obtenção dos artigos que farão parte dos estudos primários. Os estudos foram selecionados através dos critérios de inclusão e exclusão definidos previamente.

1. O indexador realizou buscas nas bases que foram previamente definidas no espaço do usuário;

2. A lista de artigos retornada foi exportada em formato Bibtex (*.bib) e importados para o aplicativo Mendeley Desktop.

3. O conjunto de artigos que foram selecionados na busca, obedeceram os critérios de inclusão e exclusão configurados no indexador.

4. Foram analisados os critérios de relevância através da leitura do resumo, introdução e conclusão dos artigos, classificando-os para posterior leitura completa dos mesmos.

5. A ordem de importância foi finalizada com a movimentação dos artigos que não tiveram relação direta com a pesquisa para outro diretório.

Todos os artigos que tiveram a ordem de relevância classificadas como alta, foram lidos completamente. Os demais, foram selecionados e utilizados trechos de textos de outros autores como referências da pesquisa, e posteriormente sua importação para o organizador de artigos Mendeley Desktop.

Avaliação da Qualidade dos Estudos

De acordo com os procedimentos realizados na Seção 2.4.2 sobre escolha por meio de ordem de relevância, a revisão buscou encontrar artigos que contemplem o desenvolvimento de redes de conteúdo utilizando a infraestrutura programável das redes SDN. E que esse ambiente disponha de recursos computacionais que possibilitem a integração com o ambiente em *Cloud Computing*.

2.4.3 Resultados da Revisão Sistemática

Na realização dos procedimentos para a RS, as buscas foram realizadas nas principais bases de artigos científicos e periódicos que envolvem as áreas da computação, e que os títulos, resumos e introdução tivessem ênfase em redes de computadores e sistemas distribuídos. Essas bases foram consultadas com um indexador que acessou os principais motores de busca para este fim, conforme *string* utilizada na Seção 2.4.2. Além da utilização do indexador SciVerse Scopus (FALAGAS et al., 2008) e a execução do protocolo de RS, também foram executados consultas utilizando os motores de busca da Google, especificamente o Google Scholar.

Após a execução da *string* de busca em 2.4.2, são exportados os resultados em formato específico para serem importados e posteriormente analisados no Mendeley. Durante a condução da RS, as fontes de pesquisa foram selecionadas, os estudos primários identificados e categorizados segundo critério de relevância, e filtrados segundo critérios de inclusão e exclusão. Não houve necessidade inclusão de artigos manualmente, pelo motivo das buscas externas ao sistema

SciVerse Scopus (buscas no Google Scholar) retornar duplicidade de artigos, dessa forma, a inclusão manual não resultaria alterações no protocolo da revisão sistemática. As *strings* de busca configuradas foram executadas para as máquinas de busca indicadas para as áreas da computação, como fontes do protocolo de RS, as seguintes bases foram consultadas: *ACM, Elsevier, IEEE e Sciense Direct*. A Tabela 2 exibe o detalhamento do quantitativo de artigos retornados por cada base científica.

Tabela 2 – Artigos retornados pela busca com o SciVerse Scopus

Indexador	Máquina de Busca / Excluídos	Qtd. de artigos retornados
Scopus	ACM	44
	Elsevier	9
	IEEExplore	94
	Science Direct	11
	Excluídos	8
Total		166

Seleção das Informações

A Tabela 2, apresenta os resultados obtidos através do uso do indexador SciVerse Scopus, onde após os procedimentos de inserção dos artigos, deu-se início à leitura dos mesmos seguindo a estratégia de leitura do *abstract*, introdução e conclusão dos artigos científicos que utilizaram métodos e ferramentas com potencial de utilização na pesquisa, sendo associados a sua respectiva referência. Dados um total de 166 artigos conforme a Tabela 1, 158 entraram para a lista dos que serão analisados e 8 artigos foram excluídos, por não atenderam as configurações exigidas no protocolo de RS .

O próximo passo foi qualificar os artigos seguindo o critério de Alta, Média ou Baixa Relevância, utilizando o botão de marcação "favorito"presentes na ferramenta. Os artigos que atenderam a todos os requisitos do protocolo da RS, são classificados como Alta Relevância e são lidos completamente de uma-a-um formando a base científica e relacional necessárias para a construção da dissertação. Dessa forma, 33 artigos foram considerados de alta relevância para construção do trabalho. Por fim, 8 artigos foram excluídos, porque tratavam-se de referências com abordagens não pertinentes a pesquisa, e referências que não puderam ser acessadas utilizando o mecanismo de busca SciVerse Scopus. Mesmo com esse *gap*, esses artigos retornaram da pesquisa pela string de busca. A Tabela 3 traz uma relação de 10 dos 33 artigos de maior relevância para fomentar a proposta de dissertação.

Tabela 3 – Relação de artigos de alta relevância

Autor	Objetivo das Propostas
Xiulei et al. (2016)	Neste trabalho foi proposto um novo framework chamado SDICN, que propõe o deploy de uma rede ICN sobre SDN com o objetivo de reduzir o caching entre os nós da rede.
Vahlenkamp et al. (2013)	Demonstra como habilitar uma rede ICN em redes IP convencionais utilizando SDN.
Hakiri e Gokhale (2016)	Propõe um *middleware* chamado POSEIDON que através da configuração e dos estados do ambiente obtidos pelo controlador, criam novas rotas, gerenciam o estado das conexões e trocam informações entre membros através de VPN.
Son et al. (2016)	Neste artigo é proposto uma arquitetura Content Centric Network (CCN) através de SDN para reduzir a quantidade de *broadcast* na rede para diminuição do tráfego na rede.
Ren et al. (2016)	Neste trabalho é proposto um gerenciamento da mobilidade de redes de conteúdo através de redes definidas por software.
Trajano e Fernandez (2016)	Este trabalho apresenta o ContentSDN. Uma arquitetura que integra ICN e SDN, através do cache de um proxy transparent para entrega de conteúdo.
Aubry, Silverston e Chrisment (2015)	Esse artigo propõe um modelo de roteamento baseado em SDN para redes CCN, que não utiliza o esquema de endereçamento convencional (por meio de IP) e nem o protocolo de comunicação OpenFlow. Foi implementado no simulador NS-3 e seus resultados melhoraram o desempenho da rede.
Sun et al. (2014)	Nesta proposta, o trabalho propõe a adaptação do protocolo OpenFlow na construção de uma OpenVSwitch híbrida para gerenciamento autonômico da rede.
Zhang e Li (2016)	Neste trabalho é proposto um novo framework para integração de SDN com ambiente *in-cloud*.
Shamugam et al. (2016)	Neste trabalho demonstra como mitigar problemas de ambientes SDN com orquestração através do OpenStack.

2.5 Trabalhos Relacionados

Nesta Seção, apresentamos os trabalhos relacionados, de acordo com as principais arquiteturas utilizadas para ICNs, bem como, os trabalhos que integram os dois paradigmas de acordo a Seção 2.2.2, no trabalho de Mougy (2015) *apud* Koponen et al. (2007). Nessa ótica, temos que a Internet dos dias atuais é bem diferente de algumas décadas atrás. A estrutura das redes ainda permanece a mesma, mas a quantidade de dispositivos que são conectados aos ambientes são exponencialmente maiores.

2.5.1 Framework ICN baseado em SDN – SDICN

Com o objetivo de demonstrar a viabilidade das redes SDN para flexibilização do novo paradigma de distribuição de conteúdo através da Information Centric Network (ICN), o artigo

de Xiulei et al. (2016) propõe um *framework* chamado SDICN. A proposta é disponibilizar um modelo de rede ICN baseado na programabilidade e nas funções de virtualização das redes SDN. O protótipo é baseado em *OpenFlow* e faz uma comparação com tecnologias Content Centric Network (CCN) (TRAJANO; FERNANDEZ, 2016), fornecendo uma proposta de *framework* que possuí centro de dados distribuídos pela rede e utiliza 3 (três) algoritmos: o Content Locating (CL), com função de escolher o melhor centro de dados para entrega; o Content Optimal Deployment (COD), que decide quais conteúdos são movidos ou copiados entre destinos, baseados em regras avançadas; e o Path Optimizing (PO), que balanceia o tráfego de acordo com a utilização do *link*.

2.5.2 Habilitando ICN sobre IP

O trabalho de Vahlenkamp et al. (2013), demonstra como habilitar uma rede ICN em redes IP convencionais utilizando SDN. Os autores descrevem que o modelo empregado neste trabalho, utiliza a pilha de protocolos convencional apenas como infraestrutura para os elementos da rede que são controlados por SDN. Descrevem que as requisições a conteúdos são realizados independente da pilha TCP/IP, que em ICN, essa possibilidade é possível pela implantação de um serviço de resolução de nomes, denominado Name Resolution System (NRS), fazendo o mapeamento dos conteúdo nomeados, descritos como Named Data Objects (NDOs), e que devem publicados na rede anteriormente. Assim, uma rede ICN pode ser dinamicamente adaptável, tal como: suportar agregação de requisições, balanceamento de carga, engenharia de tráfego e armazenamento de cache para os conteúdos mais solicitados.

Nesta mesma linha de pesquisa, no estudo de Pursuit (2016), foi desenvolvida uma arquitetura chamada Publish/Subscribe Internet Technology (PURSUIT) que funciona da seguinte forma: três nós são introduzidos em cada sistema autônomo (SA): o Redezvous Node (RN) recebe e armazena os dados das publicações que foram requisitadas, o Topology Manager (TM) estabelece rotas entre publicadores e endereçadores, enquanto o Forwarding None (FN) atual no controle da performance do encaminhamento e do cache.

2.5.3 Sistemas de *Middleware* Escaláveis

O gerenciamento da rede é realizado pelo controlador, logo, alguns autores apontam para a necessidade do controlador contatar todos *switches* entre origem e destino para estabelecimento dos fluxos de encaminhamento causando um aumento na latência do controlador. Assim, os autores Hakiri e Gokhale (2016) propõem um *middleware* para criação de rotas alternativas baseados nos estados do ambiente obtidos pelo controlador, criando novas rotas gerenciando os estados das conexões. O Proactive broker-less Subscriber Interest-Defined Overlay Networking (POSEIDON), utiliza um formato baseado em VPN para permitir que o plano de dados encaminhe esses pacotes através de tunelamento, sem a necessidade de consultas ao controlador reduzindo a latência sob o mesmo.

Além dos paradigmas relacionados na pesquisa, essas arquiteturas poderão sofrer impacto direto da Internet das Coisas (IoT), devido a alta quantidade de endereçamentos IP que irão requisitar recursos de rede e recursos de conteúdo. Assim, a proposta Scalable and Adaptative Internet Solutions (SAIL), apresentada no trabalho comparativo de Ghodsi et al. (2011) demonstra uma arquitetura com dois nós sobrepostos, sendo que um nó faz a resolução de nomes, e o outro faz o roteamento de conteúdo. Esse último ainda possuí características de cache. Neste trabalho, o autor faz uma comparação das diferentes arquiteturas para a resolução de nomes de conteúdo.

2.5.4 Sistemas de Controle de Clusters SDN

Os autores Son et al. (2016) descrevem sobre a dificuldade do mapeamento de localização de conteúdos em termos de configuração de rede, quando há aplicação em larga escala na utilização de troca de conteúdo entre requisitantes e respondentes. As requisições que são trocadas através de concentradores em nível de camada de rede, entregam as solicitações dos clientes por meio de caminhos reversos usando tabelas de interesse, Pending Interest Table (PIT) e tabelas de encaminhamento, Forwarding Information Base (FIB). A arquitetura proposta, prevê a resolução do *broadcast* entre roteadores para diminuição do consumo de recursos da rede, uma vez que o caminho reverso, pode duplicar uma resposta porque usa o mesmo caminho para várias requisições ao mesmo conteúdo.

2.5.5 Sistemas para Mobilidade de Requisições

As requisições ao mesmo conteúdo por vários clientes são discutidos no trabalho Ren et al. (2016). Os autores propõem a utilização de SDN com suas funções de desacoplamento do controle dos concentradores no projeto SDC-CCN. Segundos os autores, uma grande quantidade de requisições ao mesmo conteúdo vindas de várias direções (*handoff*) gerando *overhead* sobre a rede. Então, para resolver o problema desse *overhead* na leitura/escrita das tabelas de roteamento, os autores inserem controlador a interceptação das requisições, chamado de *software-defined controller (SDC)* para reescrever novas tabelas de roteamento e enviar as rotas para os concentradores mais significativos.

2.5.6 Sistemas com utilização de Proxies para Requisições

Da mesma maneira das arquiteturas descritas nas seções anteriores, o trabalho dos autores Trajano e Fernandez (2016), apresentam o ContentSDN. Esta arquitetura integra SDN/ICN de modo a evitar que as requisições HTTP sejam controladas estaticamente. Todas as requisições HTTP são encaminhadas para o proxy através de fluxos instalados nos *switches*. Quando uma requisição alcança o proxy, o mesmo consulta o controlador para resolver o destino final da requisição. Caso o conteúdo não esteja em cache, o proxy irá obter o conteúdo através de uma requisição fim a fim e armazenará o cache para futuras consultas. Neste trabalho, os autores separam o proxy através de dois componentes: o ContentSDN Proxy e o ContentSDN Cache,

onde esses proxies trocam informações por meio de tabelas de *hash* distribuídos, do inglês, Distributed Hash Table (DTH) minimizando o congestionamento da rede.

2.5.7 Sistemas non-IP Roteamento de Requisições

Além da proposta de ICNs como novo modelo de entrega de conteúdo na web; Da integração com redes programáveis para viabilizar uma infraestrutura flexível para ICNs, alguns autores apontam para arquiteturas que são propostas pela utilização de *in-network cache*, ao invés de analisar também os esquemas de roteamento dessas requisições. Dessa forma, o trabalho de Aubry, Silverston e Chrisment (2015) propõem um estudo mais aprofundado nos esquemas de roteamento, visto que redes CCN não utilizam o modelo TCP/IP, logo, os conteúdos são distribuídos pelo caminho através de *flooding* na rede, causando *overhead* rede, desperdício de recursos e redução de performance.

O trabalho utiliza o paradigma SDN para propor o SRSC, um esquema de roteamento SDN para redes CCN. Nesta arquitetura, os autores não objetivam a utilizam de redes IP para efetuar os esquemas de roteamento. Ao invés disso, utilizam múltiplos controladores interligados, com múltiplos domínios CCN, multiplicando as tabelas de roteamento mantendo o sistema autônomo sempre atualizado sobre a localização dos clientes da rede, evitando o envio de *floodings* na rede.

2.5.8 Sistemas Autonômicos para Redes CCN

Mesmo com a utilização de redes CCN e esquenas *in-network*, autores como Sun et al. (2014) também descrevem problemas sobre a falta de otimização de recursos da rede, pelo envio de *floods* para nomeação de arquivos mesmo em redes que não seja TCP/IP. Uma das questões apontadas, é o alto consumo de recursos de troca de conteúdos *peer-to-peer (P2P)*. Os autores descrevem algumas limitações no gerenciamento pela não atribuição de um peso justo para os fluxos do *OpenFlow*, não obtendo colaboração nó a nó da rede. Sendo assim, eles propõem utilização de OpenFlow/SDN e um CCN-Router chamado de SDN-based Intellingent Traffic Management (SITM). Através do esquema de monitoramento de *switches* do plano de dados, o controlador será capaz de definir contextos, autoanalise e definição de políticas para alcançar a inteligência na entrega de conteúdos pela rede, utilizando o gerenciamento de filas de tamanho de pacotes, gerenciamento do envio do pacote e alocação de banda para envio do pacote. Que os autores chamam de *Queue Management* e *Queue Scheduling*.

Para integrar ICN em SDN, algumas propostas são apresentadas como no estudo de Luo et al. (2014). A arquitetura chamada de Couple service Location and Inter-domain Router (CoLoR) propõe uma integração eficiente do ICN e SDN, reduzindo a configuração de fluxos OpenFlow, número de requisições de fluxos e número de entradas de fluxo no OpenVSwitch. Essa arquitetura também assume que o novo contexto de internet é baseado em domínios, e cada domínio negocia com outros domínios mapeando rotas de menor custo.

2.5.9 Sistemas de Integração com Ambientes em Nuvem

Além da distribuição de requisições e utilização de caches massivamente sob os nodos da rede, as possibilidade de integração de SDN/ICN ainda permitem sua utilização em sistemas em nuvem. O trabalho dos autores Zhang e Li (2016), apresentam um *framework* em SDN que possibilita essa integração. Os autores utilizaram o sistema de tradução de endereços (NAT) nativo do sistema *hypervisor* sem a necessidade de utilização do protocolo OpenFlow ou esquemas de tunelamento por meio de VxLANs. Os *hosts* são conectados em redes virtuais e gerenciados através de duas interfaces de rede. Em um desses adaptadores é configurado uma NAT para que a rede virtual possa alcançar a Internet, ou seja, interfaces diretamente conectadas o *hypervisor* conhecerá todas as rotas, assim regras de NAT poderão ser aplicadas, conectados duas "ilhas" de rede por meio da Internet.

Neste mesmo formato, Shamugam et al. (2016), demonstra um estudo de caso sobre as possibilidade da orquestração de ambientes SDN com controladores implantados em nuvens *OpenStack*. O trabalho apresenta varias possibilidades de eventos gerenciados por um controlador SDN, tal como, coleta de informação e decisões de encaminhamento de pacotes estarem centralizados no *openstack*, com o plano de dados mantido em um *pool* de computação. Os autores ainda descrevem sobre as possibilidades de utilização de IPs flutuantes através do *Virtual Router Redundancy Protocol (VRRP)* que é suportado pelos controladores SDN.

Por fim, para prover um ambiente confiável e escalável, de maneira que tanto a flexibilidade de topologias e a programabilidade dos *switches* estejam centralizados sob o domínio de um administrador de rede, o artigo de Nikbazm, Dashtbani e Ahmadi (2015) demonstra como interligar esse sistema e utilizar recursos de computação na nuvem. O trabalho apresenta a configuração de um ambiente SDN em *cloud* para mitigar problemas de incompatibilidade dos diferentes tipos de *hardware*, melhorando a performance e o QoS, simplificando o gerenciamento das redes. Neste trabalho, o controlador SDN é utilizado para controlar a infraestrutura do ambiente em *cloud computing*. Na Tabela 4 a coluna *in-cloud* representa a contribuição que à arquitetura proposta deverá atender.

2.5.10 Análise Comparativa

O que foi percebido durante a revisão sistemática, é que os vários autores desenvolveram os *frameworks* para atender as demandas de pesquisa a conteúdos em ICNs. Os pesquisadores utilizaram em sua grande maioria arquiteturas sobrepostas no ambiente, ou seja, informações que circulam sobre a pilha TCP/IP, como uma camada extra de acordo com os autores Vahlenkamp et al. (2013), Trajano e Fernandez (2016) e Pursuit (2016). Esse tipo de configuração é bastante utilizado, uma vez que, o paradigma ICN permite tal utilização. Os autores descrevem a utilização de caches e de sua otimização, mas não descrevem claramente, as formas de cache utilizado, apenas que são distribuídos pela rede.

Uma outra percepção é que os dados circulam de um nó para o outro, alguns por meio

dos seus sistemas autônomos, mais que não há mais uma camada de persistência para essas informações e dados, além do conteúdo inserido em todos os nós daquele segmento de rede. Essa característica alerta para o fato de que uma rota poderá ficar indisponível, o que provoca indícios de vieses com relação a replicação de conteúdo entre os nós do segmento. O trabalho dos autores Hakiri e Gokhale (2016) e Ghodsi et al. (2011) direcionam para este sentido. Ainda o estudo de Ghodsi et al. (2011), sinaliza para a resolução de nomes, mais sem se referir quais os mecanismos são mais utilizados em pesquisas sobre ICNs. Sendo assim, se os conteúdos não puderem ser recuperados em uma outra rota, a obtenção poderá ter um maior custo de roteamento, acarretando um maior consumo de recursos computacionais e de rede.

A requisição duplicada pelo mesmo segmento rede, tornaram-se um dos maiores desafios das comunidades de pesquisas na integração de redes programáveis com redes de conteúdo. Um vez que o SDN centraliza o gerenciamento, ainda resta otimizar as requisições que circulam por meio dos concentradores de rede. Assim, o estudo de Aubry, Silverston e Chrisment (2015), utiliza um contexto baseado puramente em redes non-IP, evitando *floods* na rede, mas não descreve sobre o mecanismo de cache utilizado, apenas que faz cache em memória. Isso poderá tornar-se um gargalo para o sistema *hypervisor*; também não há evidências claras sobre o controle da fila de requisições, bem como sobre o roteamento baseado em nomes de conteúdo.

O controle de requisições, pode ser visto nos trabalhos de Sun et al. (2014), que através do SDN-based Intellingent Traffic Management (SITM) estabelece rotas alternativas de menor custo. E no trabalho de Luo et al. (2014), que através da arquitetura Couple service Location and Inter-domain Router (CoLoR), monitora quantidade de fluxos no *switches*. Mas é sabido que nem sempre uma rota de menor custo é o melhor caminho para entregar um pacote, o que leva a hipóteses sobre como o sistema executa esse negociação entre domínios, através de tuneis ou de proxies. Entendemos que a utilização de mecanismos para gerenciamento da fila (*Queue Management*) talvez esclareçam acerca dessa otimização de QoS.

Por fim, Zhang e Li (2016) demonstra a utilização da capacidade de tradução de nomes de adaptadores de rede. Acreditamos ser uma boa alternativa, visto que, na criação de *switches openflow* as portas virtuais dos concentradores são capazes de se comunicar com os adaptadores físicos dos *hypervidores*. Essa é uma das características apontadas por Nikbazm, Dashtbani e Ahmadi (2015), a possibilidade de termos ambientes SDN completos na nuvem, sobrepostos em redes TCP/IP com endereços traduzidos ou interligando por meios de IPs flutuantes de roteadores redundantes. A Tabela 4, apresenta um resumo da utilização de recursos pelos autores para integração de SDN/ICN, comparando com a proposta desta pesquisa.

Tabela 4 – Comparativo das tecnologias utilizadas nos trabalhos relacionados

Infraestrutura			**Arquitetura ICN**					
Autor	**Framework**	**Integração SDN/ICN**	**DONA**	**CCN**	**PSIRP**	**SAIL**	**Netinf**	***in-cloud***
Ghodsi et al. (2011)	COCONET	✓		✓	✓			
Vahlenkamp et al. (2013)	ICN/IP	✓	✓					
Luo et al. (2014)	CoLor	✓	✓					
Sun et al. (2014)	SITM	✓		✓	✓			
Nikbazm, Dashtbani e Ahmadi (2015)	ODL	✓					✓	✓
Aubry, Silverston e Chrisment (2015)	SRSC	✓	✓		✓			✓
Xiulei et al. (2016)	NOX	✓	✓		✓			
Trajano e Fernandez (2016)	SDICN	✓		✓				
Pursuit (2016)	ICN/IP	✓	✓		✓			
Hakiri e Gokhale (2016)	POISEDON	✓	✓		✓		✓	
Son et al. (2016)	SDN-Based CCN			✓				
Trajano e Fernandez (2016)	ContentSDN	✓	✓				✓	
Zhang e Li (2016)	SDN NAT						✓	✓
Shamugam et al. (2016)	OpenStack VRRP						✓	✓
Esta proposta de Pesquisa	**Ryu OS**	✓	✓		✓		✓	✓

3 Metodologia e Técnica de Pesquisa

A natureza deste trabalho de pesquisa é de caráter bibliográfico, porque está baseado em material já elaborado, constituído principalmente de artigos científicos, livros, jornais, revistas, teses e dissertações (MARCONI; LAKATOS, 2003). Com base em seus objetivos está classificada como explicativa, pois tem como meta identificar os fatores que determinam ou que contribuem para ocorrência dos fenômenos. Quando aos procedimentos, é considerada experimental, porque busca através da replicação do cache em nuvem determinar sua viabilidade, selecionando formas para validar redes conteúdo sob SDN (GIL, 2002).

3.1 Métodos de Pesquisa

Ainda de acordo Gil (2002), a pesquisa é considerada experimental quando dado um objeto de estudo determinado, escolhem-se variáveis independentes capazes de influenciar variáveis dependentes. Dessa forma o pesquisador provoca alterações no ambiente observando se a inferência aplicada produziu algum resultado esperado. Sendo assim, essa pesquisa pode ser considerada experimental, porque objetiva validar a redução do consumo da banda, diminuição da quantidade de processamento de requisições, tempo de roteamento para o cache e replicação em nuvem, mediante utilização de microsserviços instanciados pelo controlador central da rede programável.

Para alcançar os objetivos desta pesquisa experimental, foram necessárias as seguintes atividades:

1. Revisão da Literatura;
2. Seleção das ferramentas de *hardware* e de *software* necessárias para composição do sistema e desenvolvimento do protótipo;
3. Escolha da(s) topologias que serão utilizadas e dos protocolos de comunicação do sistema;
4. Planejamento do Datacenter Local integrado com a nuvem;
5. Desenvolvimento dos microsserviços para os nodos da rede;
6. Execução dos experimentos;
7. Análise e interpretação dos dados coletados.

3.1.1 Revisão da Literatura

Uma pesquisa bibliográfica é um apanhado geral sobre os principais trabalhos já realizados, que tenham suma importância e sejam capazes de fornecer dados atualizados e relevantes, e que estejam relacionados ao tema da pesquisa (MARCONI; LAKATOS, 2003). De acordo com essa conceituação a revisão da literatura que aborda o tema escolhido tem como objetivo suportar os conceitos envolvidos sobre redes programáveis, redes de conteúdo e suas relações com o ambiente em nuvem. A revisão da literatura para esta pesquisa foi realizada por meio de revisão sistemática apresentado na Seção 2.4.

3.1.2 Seleção de conjunto de *Hardware* e *Software*

Nesta atividade, uma pesquisa sobre qual o *hardware* necessário para suportar o desenvolvimento da arquitetura que permita que o administrador de rede possua completo acesso ao sistema com todas as permissões possíveis, que seja capaz de suportar uma rede SDN, possua dispositivos de armazenamento físico e tenha permissão para realizar conexão com redes internas e externas. A ferramenta escolhida deverá atender aos requisitos elencados e ainda deve suportar o uso de linguagens de programação de alto baixo e alto nível, e ser possível sua utilização na programação de redes e sistemas operacionais.

Nesta atividade, um pesquisa sobre qual o *software* necessário para implementar a arquitetura. Basicamente, o software para rede deverá esta apto a ser escrito, salvo e compilado no próprio *hardware* que ficará funcionando em execução. O *software*, também deve ser capaz de possuir funções de rede e de escalonamento de processos, para no caso da necessidade de se ter várias instância de *software* funcionando sob o mesmo *hardware*.

3.1.3 Seleção da Topologia e dos Protocolos de Comunicação

Essa atividade definirá qual a melhor topologia que será utilizada para atingimento dos objetivos da pesquisa. A princípio, o sistema deverá ser capaz de operar sob qualquer topologia de rede, mais poderá ser definido uma topologia geral que atenda o propósito da pesquisa. Um outro ponto, é que a topologia escolhida deverá permitir a conexão com redes externas, disponibilizando a possibilidade de termos redes heterogêneas funcionado como se fossem redes LANs.

Neste ponto da pesquisa, protocolos de comunicação padronizados mundialmente deverão funcionar plenamente nas instâncias dos microsserviços quanto no *hardware* que suportará o sistema. Esses protocolos, deverão permitir a resolução de nomes e endereços aos domínios específicos e elementos da rede.

3.1.4 Planejamento do Datacenter Programável Integrado com a Nuvem

Esta etapa da pesquisa, será definido como os experimentos serão conduzidos, e quais as etapas serão necessárias para o funcionamento dos elementos da arquitetura. Ainda nessa etapa, serão definidos quais os cenários dos experimentos, e quais itens que serão utilizados como entradas para o sistema, e quais variáveis serão analisadas, qual será o tempo de execução do experimento e quantos ciclos de execução serão executados. Os dados coletados serão armazenados em alguma tecnologia de armazenamento confiável para serem analisados.

Ainda nesta atividade, uma análise sobre qual a tecnologia será utilizada para integrar a arquitetura com a nuvem. Serão estudados e implementadas tecnologias que caracterizam-se pela abertura de tuneis utilizando a infraestrutura de internet. Serão estudados como os nodos da arquitetura terão acesso ao enlace, bem como, a redundância desse acesso para o caso de falhas. Também serão definidos as necessidades de utilização de balanceadores de carga na execução dos ciclos de testes. Com base na diagramação da topologia, o protótipo passa a ser implementado.

3.1.5 Desenvolvimento dos Microsserviços

Nesta atividade, serão desenvolvidos os serviços de conteúdo que irão compor a arquitetura. Serão implementados serviços de conteúdo que armazenem as informações requisitadas, definindo quais tipos de informação vão ser acessadas, e como essa informação será entregue aos seus requisitantes.

Ainda nessa atividade, será pesquisado as regras e papéis da nuvem para disponibilizam/acesso a esses serviços. Os serviços de conteúdo que estarão na nuvem deverão ser acessados localmente e remotamente de qualquer nodo da rede, apontando para a redundância do sistema. A prototipação do fluxo desses microsserviços deverá ser apresentada e o passo a passo do acesso ao conteúdo descrito.

3.1.6 Execução dos Experimentos e Análise dos Resultados

Nesta atividade, serão feitos os laboratórios necessários para obtenção dos dados. O experimento ficará em execução durante o período necessário para levantamento do conjunto de dados necessário para análise. Serão executados os mesmos procedimentos, em cenários locais e remotos. Também serão executados os mesmos procedimentos com topologias diferentes. Dessa forma, é possível obter a validação da arquitetura.

Por fim, ainda nesta atividade, a análise, interpretação e validação do conjunto de dados serão executado de acordo os resultados estatísticos obtidos através do conjunto de dados, analisando os resultados e verificando se os métodos e técnicas utilizados podem beneficiar a utilização de redes programáveis para orquestrar redes orientadas a conteúdo.

3.2 Operacionalização do Ambiente

Nesta seção será apresentado o conjunto de ferramentas utilizado para compor a base de funcionamento da arquitetura proposta na pesquisa.

3.2.1 Sistema de Virtualização

Nos sistemas de computação, virtualização significa uma abstração de componentes físicos em objetos lógicos gerenciados por meio de um hospedeiro. A criação de objetos virtualizados permitem que infraestruturas inteiras de serviços sejam disponibilizados. Por exemplo, criação de *Virtual Local Area Networks* (VLANs), abstraindo em forma de objeto concentradores e roteadores que possuem essa função podem ser virtualizados via *software* e terem um desempenho tão quanto igual ou melhor do que equipamentos físicos. Além do que, esses *switches* virtuais poderão ser facilmente gerenciados, diminuindo a complexidade de sistemas e configurações para administradores de redes.

A virtualização de componentes de *hardware* permite da mesma forma utilizar objetos lógicos para compor sistema de armazenamento, como, *Storage Area Network* SAS, com alta performance e flexibilizando o espaço em disco virtual por meio da virtualização de *Filesystems* em bloco flexíveis que são facilmente manipulados. A Figura 17 mostra um modelo simples de sistema de virtualização.

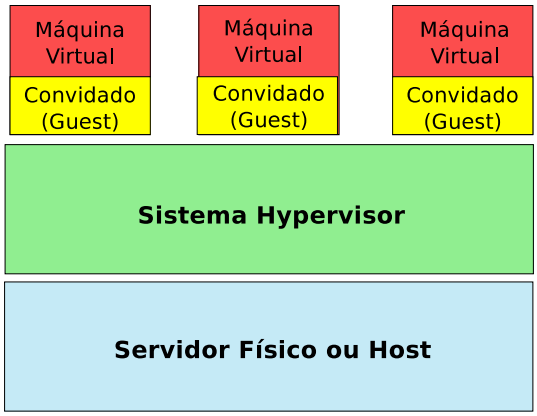


Figura 17 – Sistema de Virtualização Básico

Para esta pesquisa optou-se por sistemas de virtualização que tivessem licença livre e possuíssem código *OpenSource*. Além disso, o sistema de virtualização utilizado na pesquisa satisfaz as três propriedades básicas segundo Portnoy (2012):

- **Fidelidade**: O contêiner deve criar as Maquinas Virtuais (VMs) essencialmente idênticas a máquinas físicas, ou seja, deverá representar o *hardware* dessas máquinas com todos seus recursos.
- **Isolamento**: O contêiner deve ter pleno controle sob os recursos utilizados pelos *Guests*.

- **Performance**: Não deve ter desempenho inferior ao do hospedeiro do qual faz parte.

3.2.1.1 Sistema Hypervisor do Tipo I

Para compor o ambiente em *cloud computing* optou-se pela utilização de *hypervisor* do tipo I. Esse modelo é conhecido como "*hypervisor nativo*"ou "*bare metal*", isso significa que seu funcionamento opera diretamente sobre o *hardware*, sem intervenção de uma camada de sistema operacional. Além do mais, essa característica permite que uma VM se comunique diretamente com o *hardware* hospedeiro melhorando o desempenho para as VMs.

A motivação para utilização de modelo de virtualização tipo I para compor o ambiente em nuvem, vem do fato da obtenção de uma instância de máquina com amplo processamento e 8 Gigabytes de memória RAM e 100 Gigabytes de armazenamento em disco para instalação do sistema operacional convidado. Neste caso, optou-se em utilizar o Ubuntu Server 16.04.5 LTS com o virtual *hardware* mencionado e 04 (quatro) adaptadores de rede.

Além disso, o ambiente Local pode ser inserido em uma virtualização do Tipo II, facilitando a mobilidade para execução do experimentos, uma vez que, o sistema instalado em computadores móveis, poderão facilitar a conexão com o ambiente na nuvem de qualquer lugar.

3.2.1.2 Sistema Hypervisor do Tipo II

Na composição do ambiente *Local Area Network* (LAN) optou-se pela utilização de *hypervisor* do tipo II. Hypervisores do Tipo II tratam-se um *software* (contêiner) que funciona sobre um sistema computacional tradicional, ou seja, é uma aplicação instalada em um sistema operacional. Os benefícios que podem ser descritos para utilização desse modelo são:

- Suporte para uma grande variedade de *hardware*.
- Facilidade de instalação e implantação. Um vez que o trabalho configuração de *hardware* como rede e armazenamento já é suportado pelo sistema operacional que mantêm o contêiner.
- Mobilidade. Para essa pesquisa houve a necessidade de mobilidade, já que *hypervisor* tipo II, podem ser instalados em PCs e Laptops com facilidade. Isso permite o acesso a nuvem de varias formas, e o tunelamento dos controladores SDN com criação de adaptadores de rede virtuais.

A Figura 18, mostra a taxonomia dos sistemas de virtualização utilizados para hospedar os SOs convidados (*Guests*) suportando a implementação da rede programável. Na rede LAN é utilizado Ubuntu Server 16.04.5 LTS com o virtual *hardware* descrito anteriormente, e mais 04 (quatro) adaptadores de rede, memória e disco (HD). Na camada inferior temos o *hardware* físico para instalação do gerenciador. Acima temos o gerenciador (do inglês *Virtual Machine Monitor*

(VMM)), podendo ser *bare metal* [1] para os do tipo I, ou *Host Operating System* para os do tipo II. Nas camadas acima, os sistemas operacionais convidados (*Guest OS*) para as aplicações que funcionarem sob os mesmos.

Figura 18 – Tipos de Hypervisores, adaptado de (DESAI et al., 2013)

3.2.2 Plataforma de Emulação de Redes SDN MININET

Nesta pesquisa, a motivação para uso de recursos que compõe o protótipo contempla a utilização de ferramentas que possuem licenças gratuitas, bem como possuam código fonte aberto, i.e., *open source*. Logo, para prover a base para a implementação da rede virtualizada, foi utilizado o Mininet Emulator (OLIVEIRA et al., 2014). O protocolo OpenFlow já se encontra devidamente padronizado de acordo com a RFC7426 (HALEPLIDIS et al., 2015). Entretanto, as redes programáveis ainda não estão, mesmo com apoio de alguns fabricantes, o projeto para a padronização ainda está sendo definido.

Para tanto, foi escolhido o uso de emulador para criação dos sistemas de rede que compõe o objeto dessa pesquisa. Através de uso de emuladores, podemos reduzir custos com a pesquisa, porque executamos simulações com custo baixo, flexibilidade, controlabilidade e escalabilidade de uma forma dinâmica e eficaz. Dessa forma foi possível executar programas e operações de dispositivos reais realizando a iteração entre eles.

O Emulador Mininet (LANTZ; HELLER; MCKEOWN, 2010) cria *hosts* virtuais baseado no método de processos virtualizados e um mecanismo de espaço de nomes que são suportados pelo Kernel Linux nas versões 2.2.26 acima. Esse método permite a separação das interfaces de rede, tabelas de roteamento de diferentes *hosts*. As *switches* virtuais no emulador Mininet são do tipo *software* Openflow *switch* conhecidas como: Open vSwtich. Neste protótipo foi utilizado a versão OpenVSwitch 1.3 (PFAFF et al., 2009).

3.2.2.1 Elementos de emulação Mininet SDN

O Mininet cria elementos de rede, customizando-os, compartilhando esses elementos com redes do ambiente real, por exemplo "*bare-metal network*"executando interações entre eles. Esses elementos podem ser descritos da seguinte forma: *Hosts, Switches, Controllers e Links*.

[1] ***Bare Metal*:** É um termo utilizado quando sistemas são instalandos diretamente sobre o *hardware* que ocupam, ou seja, a primeira camada acima do *hardware*

Um *host* é um processo com sua interface de rede executando no sistema operacional. Cada processo possui uma interface de rede própria, portas, endereçamento e tabelas de roteamento (ARP e IP). Todos esses processos são disponibilizados no *kernelspace* e no *namespace* de modo que, através do *kernel* haja comunicação entre eles, bem como entre os processos da rede do hospedeiro.

A criação do ambiente simulado mínimo cria um controlador padrão, denominado *Controller*, onde através dele é possível criar *links* entre os elementos virtuais através de suas interfaces de rede virtuais. Dessa forma um ambiente mínimo pode ser criado com o comando:

$: *mn –topo single,3 –mac –switch ovsk –controller remote*

Este comando cria uma 3 *hosts* virtuais, conectados ao *switch* OpenFlow com 3 portas por meio de *links* virtuais, definidos MAC e endereços IP para cada *host*, configurando os fluxos entre os elementos para o controlador remoto (*remote controller*). Neste exemplo o controlador está executando localmente no mesmo sistema hospedeiro que guarda o S.O convidado. A Figura 19 mostra a topologia *single*. Nesta topologia todos os elementos estão conectados ao mesmo concentrador.

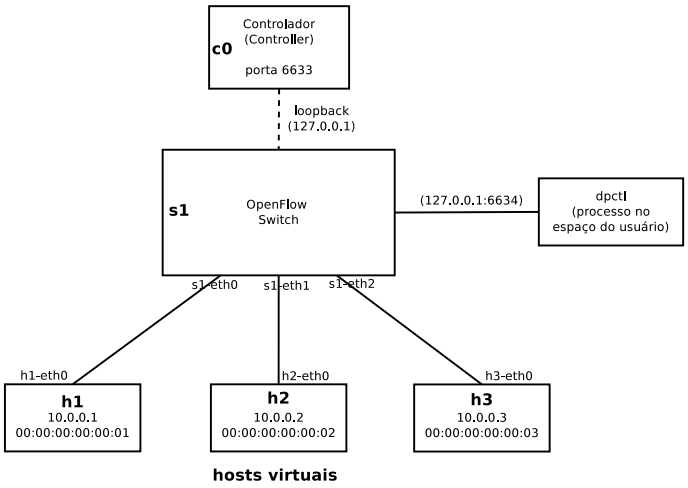


Figura 19 – Simulação de Topologia Básica (*Single*)

3.2.2.2 Topologias do Mininet Emulator

Mininet pode executar topologias padrão que podem ser descritas como: *Minimal*, *Single*, *Reversed*, *Linear* e *Tree* (KAUR; SINGH; GHUMMAN, 2014). É essencial entender o método de nomeação das interfaces de rede de todos os elementos para utilizar a ferramenta, uma vez que, os nomes dessas interfaces são utilizados para configurar o *backend* do protótipo da pesquisa. Os *swithces* podem ser nomeados de s1 até s(N). *Hosts* são nomeados de h1 até h(N). Suas interfaces são nomeadas de acordo com o nomes do *host* na rede, ou seja, se o *host* tem prefixo "h1", sua interface de rede será nomeada como "h1-eth0". Os *swtiches* seguem o mesmo princípio, com um diferencial, suas interfaces iniciam em "1", logo se um *swtich* tem prefixo em "s10", sua

interface de rede inicial será nomeada de "s10-eth1". A Figura 20 apresenta três das topologias que utilizam um único Open vSwitch.

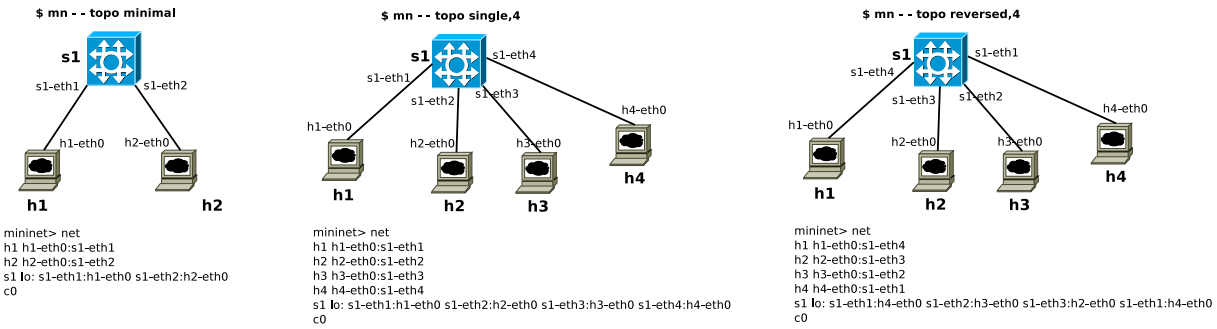


Figura 20 – Topologias Minimal, Single e Reversed.

A topologia Minimal, compõe a rede programável mais simples, com 1 *OpenFlow swtich* e 2 *hosts*, e os *links* configurados entre eles. Na topologia Single, podem ser criados *hosts* que variam de 1 a k *hosts*, definindo a comunicação entre eles de 1 a k *links* no *switch*. Na topologia Reversed, tem-se exatamente a mesma configuração da topologia Single, o que a torna diferente é o modo como os *links* são configurados entre os elementos. As interfaces iniciais do *openflow switch* vão se conectar de forma reversa aos *hosts*, i.e., "s10-eth1:h4-eth0 s10-eth2:h3-eth0..."de acordo com o comando "$ mininet> net".

As possibilidades para operação de redes mais complexas podem ser obtidas por meio das topologias Linear e Tree. Nesse formato, temos o modelo de redes convencional representado, e isto serviu para embasar o uso das topologias Tree no lado da LAN e Linear no *backend* da *cloud*. A Figura 21 mostra o formato Linear. Neste formato podemos ter k *hosts* e k *switches*, logo, o que caracteriza esse formato é a interligação dos *swtiches* paralelamente, semelhante a topologia barramento.

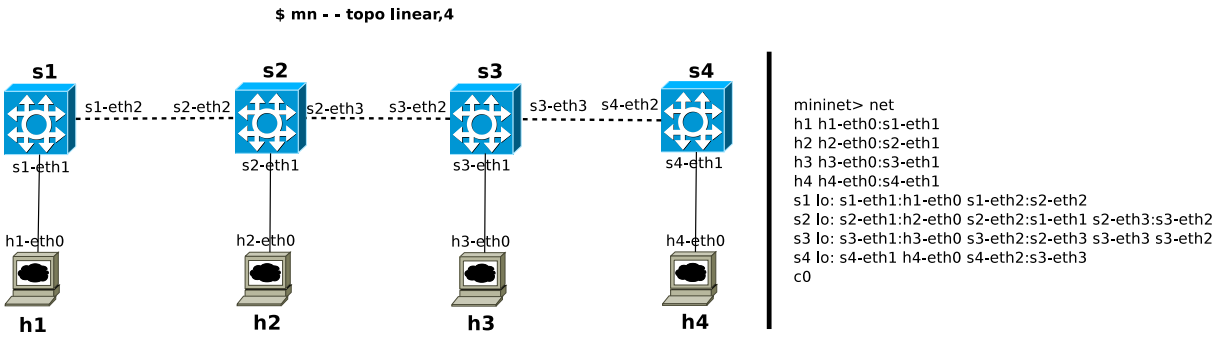


Figura 21 – Topologia Linear.

Ainda se tratando de redes complexas, o modelo Tree será representado por meio da Figura 22. Esta topologia, além de incorporar todas as características do modelo Linear, ainda possui níveis hierárquicos, ou seja, pode-se construir topologias que possuam k *hosts* e k *switches*, sendo que esses concentradores podem sem empilhados em "n"níveis. Nesta pesquisa isso serviu

para facilitar a modelagem do tráfego, termo conhecido como *Traffic Shaping* [2], permitindo que observações fossem feitas em comparação com ambientes reais.

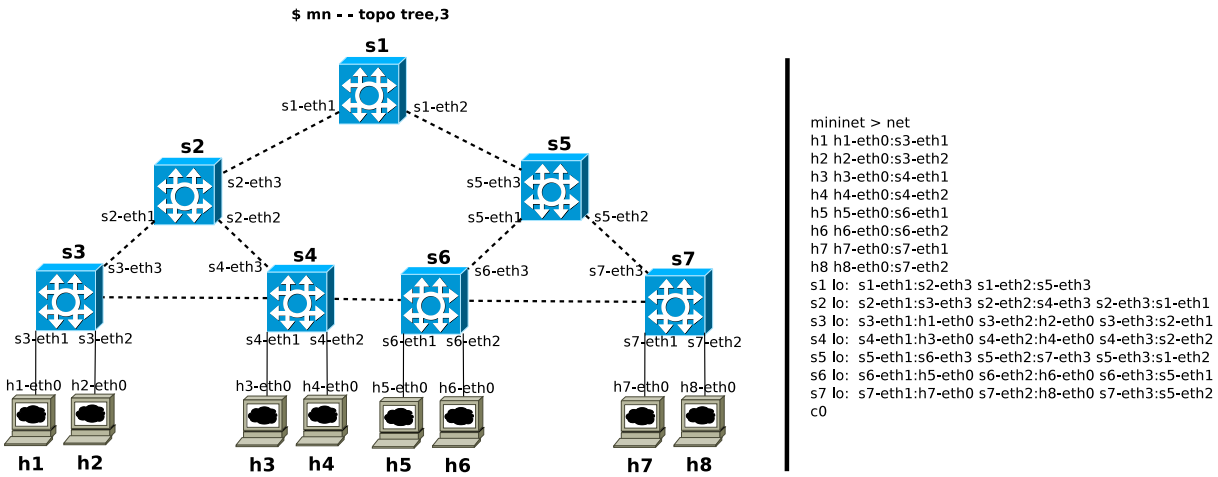


Figura 22 – Topologia Tree.

3.2.2.3 Topologias Customizadas

Além das topologias descritas na Seção 3.2.2.2, nesta pesquisa foi necessária a utilização de topologias customizadas (KAUR; SINGH; GHUMMAN, 2014), devido a necessidade de inicialização de serviços em *background*, definir vazão de *links* na Lan e para a Nuvem (*Cloud*), bem como executar o tunelamento entre a Lan e a Nuvem. Por exemplo, o Código 3.1 apresenta um pequeno código escrito em Python que possui 2 *swtiches* e 2 *hosts* com os *switches* interligados entre si, e os dois *hosts* conectados ambos cada uma em um determinado concentrador.

```
from mininet.topo import Topo
class MyTopo( Topo ):
    "Topologia Linear Basica."
    def __init__( self ):
        "Criando Topologia Personalizada."
        # Inicializando a topologia
        Topo.__init__( self )
        # Adicionando hosts e switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )
        # Adicionando links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Código 3.1 – Topologia Customizada

[2] **Traffic Shaping**: permite o controle da vazão, limitando o tráfego de acordo com regras estabelecidas para protocolos específicos (SAEED et al., 2017).

Por fim, com base na execução do Código 3.1, a Figura 23 apresenta o ambiente criado a partir de uma execução personalizada. Esse código pode ser executado a partir do seguinte comando no terminal da VM:

$: *mn –custom ~/pastaDoCode/nomeDoArquivo.py –topo mytopo*

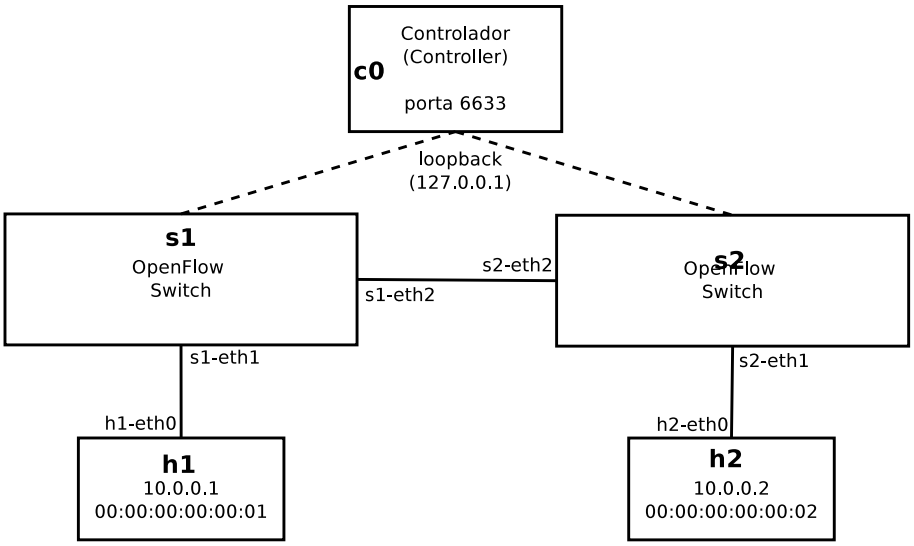


Figura 23 – Topologia Linear Customizada.

3.2.3 Resolução de Nomes de Domínio

A utilização da tradução de nomes de endereços se faz necessário para dinamizar o funcionamento do protótipo, e permitir que requisições *web* possam ocorrer sem o direcionamento direto para destinos finais. Dessa forma os endereços IPs dos destinos finais, puderam ser encapsulados pela resolução de nomes de domínio(*resolver*). Quando um serviço de nomes está presente em uma rede, nenhum *host* precisa conhecer o endereço físico de um servidor ou de um recurso que se deseja acessar, por exemplo, uma requisição *web*. Usando as informações contidas no serviço de nomes, uma requisição *web* pode procurar o endereço (ou qualquer atributo ou propriedade) por meio de um recurso interrogativo, ou seja, uma solicitação para um domínio *on-line*, onde o serviço de nomes fará a busca através do servidor de nomes. Termo conhecido como *Querying*. Assim, recursos podem ser adicionados, movidos, alterados ou deletados de um local não conhecido pelo requisitante.

Um pode ser visto como um banco de dados que mantém armazenados os nomes de *hosts*, e atribui a sua propriedade, seu endereçamento IP. No protótipo, o servidor de nomes simplificou o gerenciamento da rede, tornando as requisições dinâmicas para responder com maior facilidade as mudanças.

3.2.3.1 Funcionamento Primário

No protótipo da pesquisa foi utilizado o *Domain Name System* (DNS), globalmente distribuído, que fornece o mapeamento entre *hostnames* e endereços IP ou endereços IP para *hostnames* somente para os servidores da arquitetura, seja local ou em nuvem. Para as SDNs o método de tradução é alcançado por meio de um DNS Resolver (uma aplicação DNS conhecida em sistemas UNIX como BIND), que envia uma DNS *query* para o servidor DNS localizado no hospedeiro requisitando a informação definidas no Resource Record (RR) do sistema (CISCOSYSTEMS, 2018). Para o protótipo, utilizamos configuração padrão, apenas com *querys* Recursivas. A Figura 24 mostra o funcionamento básico de uma requisição recursiva, onde o processo interativo utilizado pelo BIND (*recursive resolver*) para responder uma mensagem (*query message*) é enviada pelo DNS Resolver, para posteriormente ser devolvida por uma *query response message* para o cliente requisitante.

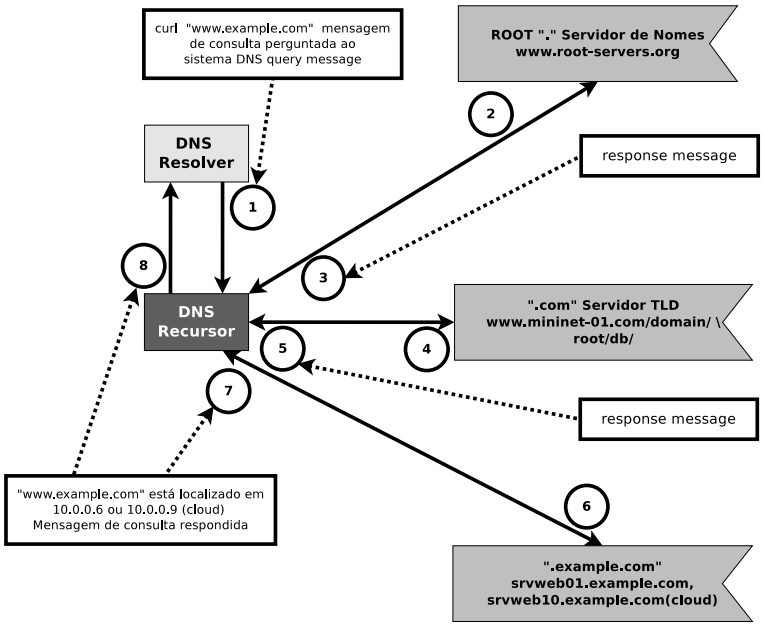


Figura 24 – Consulta recursiva ao DNS adaptado de (CISCOSYSTEMS, 2018).

1. O DNS resolver BIND) envia uma mensagem de consulta para o *recursive resolver* para saber o endereço de "www.example.com".

2. O DNS recursor envia uma mensagem de consulta para o servidor de nomes local, apontando para o espaço de nomes de domínio ".com".

3. O servidor de de nomes raiz envia uma mensagem de resposta da referência do DNS para o DNS resolver (BIND) informando para consultar o servidor de nomes principal *generic top-levels domain* (gTLD) pelo domínio ".com".

4. BIND envia uma consulta para o gTLD apontando para o domínio local "example.com".

5. gTLD envia uma mensagem de resposta com a referencia do DNS, informando para consultar os servidores "srvweb01.example.com"ou "srvweb02.example.com"sobre esse domínio.

6. BIND envia uma consulta para o "srvweb01.example.com"perguntando por "www.example.com".

7. Os servidores "*.example.com"envia uma mensagem de resposta para o BIND com o endereço IP e o nome.

8. O BIND envia uma mensagem de resposta para o DNS informando o endereço IP e o nome de "www.example.com"para que o sistema armazene em sua base para novas consultas, *Resource Record* (RR).

3.2.3.2 Operação de consulta ao DNS

Nesta Seção, será descrito a resolução de nomes de endereços de uma forma sucinta, pois não há necessidade da exploração detalhada das funções de DNS em nível de Internet. Um vez que, o protótipo utilizado para compor a pesquisa é executado em ambiente virtualizado, porém emulado. A motivação é tão somente descrever seu funcionamento no protótipo, realizando consultas (*querys*) reversas não interativas através do BIND utilizando portas aleatórias, sem necessidade de configuração em nível de um servidor DNS Nativo no sistema utilizando uma porta de serviço, por exemplo, porta de serviço número 53 configurada. Isto poderia descaraterizar o *data plane* SDN.

As operações que o serviço de resolução de nomes executa são basicamente quatro: Consultas *DNS Querys*, Mapeamento Reverso (*Reverse Mapping*), Manutenção da Zona (*Zone Maintenance*) e Segurança do DNS (*DNS Security*). Para esta pesquisa, não houve necessidade de configurações complexas para que a resolução fosse atingida, logo, foi utilizado apenas, o sistema de *querys* internas que a maioria dos sistemas operacionais modernos já possuem, sejam eles sistemas Windows, Linux, BSD ou UNIX. Assim, o sistema de resolução de nomes da arquitetura da pesquisa atende os seguintes aspectos:

- **Consultas Recursivas (*Recursive Querys*)**: atua quando o serviço de resolução (*resolver*) tem de fazer toda a verificação para o retorno da resposta completa do endereço que foi requisitado na rede, consultando o banco de dados de nomes e endereços.

- **Iterativo ou não recursivo (*nonrecursive*)**: serviço de resolução já possui a relação de nomes e endereços no arquivo de configuração, não há necessidade de solicitações ao DNS local. Sistemas operacionais modernos possuem essa característica.

- **Consulta Reversa (*Inverse Querys*)**: Tipo de consulta de ordem inversa a um RR presente no banco de dados de nomes. A consulta é feita em ordem inversa, por exemplo, através de endereços IPs. Sendo que a RFC 3425 o tornou obsoleto.

Sendo assim, essa pesquisa utilizou formato Iterativo ou Não Recursivo para todos os elementos da rede, permitindo que o proxy reverso consulte o arquivo de nomes, encapsulando todos os endereços IP da rede. A Figura 25 mostra esse serviço de resolução local com mapeamento da rede *Resource Records*(RR) em aquivo.

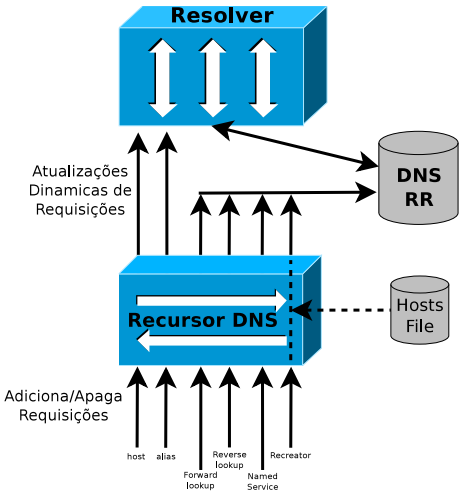


Figura 25 – Banco de dados de nomes em arquivo.

3.2.3.3 Consultas Iterativas Não Recursivas (*norecursive querys*)

No protótipo desta pesquisa, múltiplos servidores *web* disponibilizam conteúdos na ICN, dessa forma, por meio de consultas não recursivas, descartamos a necessidade de utilizarmos um servidor tipicamente "SDN DNS"para que as requisições feitas pelos elementos da rede não sejam diretamente relacionados com o endereçamento IP. Logo, para realizarmos o procedimento de resolução de nomes de endereços através de redes SDN, utilizará um sistema não recursivo para alcançar as informações por meio do alias de endereçamento dos *hosts*, realizando a resolução de nomes e endereços em sistemas SDN (vide 25). A Figura 26 mostra um exemplo de uma consulta DNS convencional.

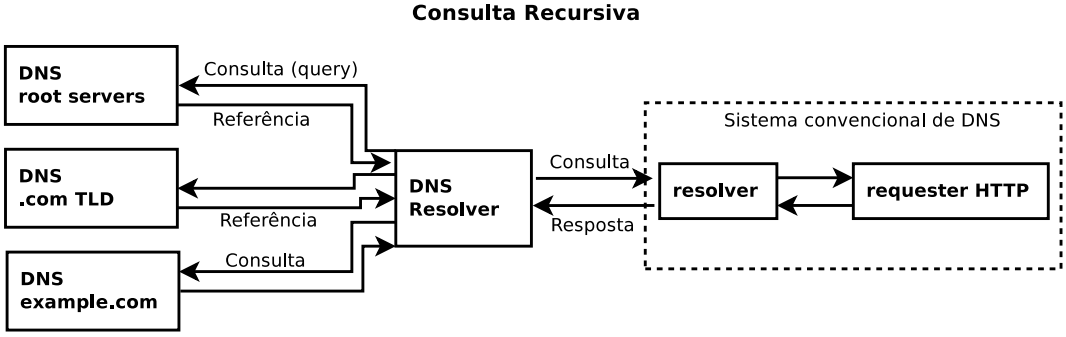


Figura 26 – Consultas Recursivas a um servidor DNS.

Todas as requisições vão ao DNS, que se encarrega de buscar a informação completa para identificação de um *hosts* na rede. Já na Figura 27, temos uma consulta interativa, quando o mapeamento é feito de forma mínima para garantir a referência do nome ao IP de algum elemento

da rede. As consultas são realizadas pelo serviço (um processo), diretamente às referências do endereçamento.

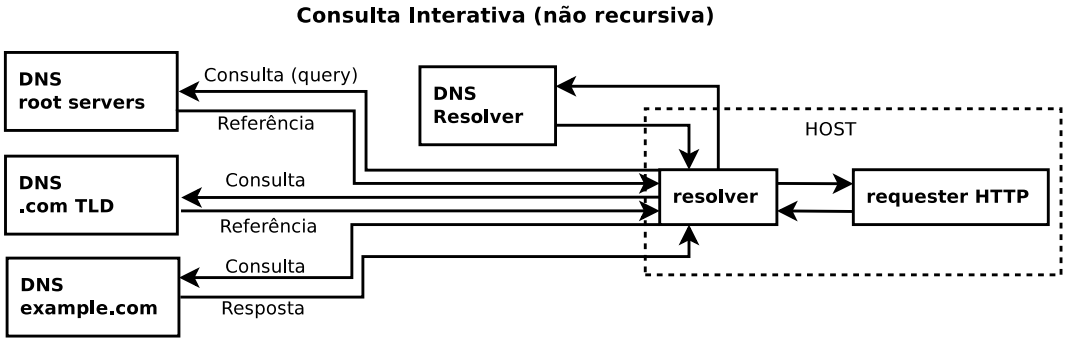


Figura 27 – Consultas Interativas não recursivas.

A Tabela 5, apresenta o mapeamento de nomes e endereços para todos os elementos de rede do protótipo deste projeto de pesquisa. Todos os controladores SDN, seja local ou na nuvem, devem conhecer todos os endereços do sistema. Acessando seus respectivos *resolvers* físicos obtendo os endereços dos elementos do sistema.

Tabela 5 – Mapeamento em Arquivo para Resolução de Nomes e Endereços

Dmínio example.com				
Endereço IP	**FQDN**	**Alias**	**Host SDN**	**Localização**
Web Servers				
10.0.0.1	srvweb01.domain.local	srvweb01	H1	Local
10.0.0.2	srvweb02.domain.local	srvweb02	H2	Local
10.0.0.5	srvweb05.domain.local	srvweb05	H3	Local
10.0.0.7	srvweb07.domain.local	srvweb07	H7	Local
10.0.0.10	srvweb10.domain.local	srvweb10	H10	Cloud
10.0.0.11	srvweb11.domain.local	srvweb11	H11	Cloud
10.0.0.12	srvweb12.domain.local	srvweb12	H12	Cloud
10.0.0.13	srvweb13.domain.local	srvweb13	H13	Cloud
Load Balance				
10.0.0.6	srvlb01.domain.local	srvlb01	H6	Local
10.0.0.9	srvlb02.domain.local	srvlb02	H9	Cloud
Registro de Domínio				
10.0.0.6	example.domain.local	example.com	H6	Local
10.0.0.9	example.domain.local	example.com	H9	Cloud
Hospedeiros				
172.16.0.22	mininet-vm01.domain.local	mininet-vm01	———	Local
172.16.0.23	mininet-vm01-domain.local	mininet-vm02	———	Cloud

3.2.4 Sistema Operacional de Rede (SON)

O sistema escolhido para controle do *data plane* da arquitetura foi Ryu SDN Framework (TOMONORI, 2013). A característica determinante para a escolha desse *framework*, foi devido todo o seu sistema ser desenvolvido em linguagem de programação de alto nível, a liguagem

Python (ROSSUM; DRAKE, 2011). Além disso, possui sua arquitetura baseada em componentes, o que flexibiliza o conjunto de aplicativos que funcionarão sob o mesmo controle (ARBETTU et al., 2016). Outras caraterísticas que foram consideradas para sua utilização estão enumeradas a seguir:

1. Plataforma OpenSource com código fonte disponível.
2. Suporta configuração de Plugins para conexão e comunicação de *hardware* e *software*.
3. Suporte a vários protocolos *SouthBound Interface*.
4. Manipulador de eventos através de REST.
5. Sem necessidade de autenticação para enviar/escrever os eventos.
6. Amplamente utilizado pela comunidade de pesquisa.

O Ryu SDN Framework, também utiliza o mesmo limite de confiança de outros controladores conhecidos nas comunidades de pesquisa, tais como: OpenDaylight (ODL) (MEDVED et al., 2014) e Open Network Operating System (ONOS) (BERDE et al., 2014). A Figura 28, mostra um Diagrama de Fluxo de Dados (DFD) de um controlador SDN. Pode-se observar uma maior relação de confiança com OpenVSwitch, e com os processos e iterações um pouco mais baixo. Isso se deve a atenuações dos ambientes ao qual estão inseridos, por exemplo, um estouro de memória RAM no hospedeiro devido uma aplicação no espaço do usuário, poderia trazer oscilações e aumentar a latência dos controladores.

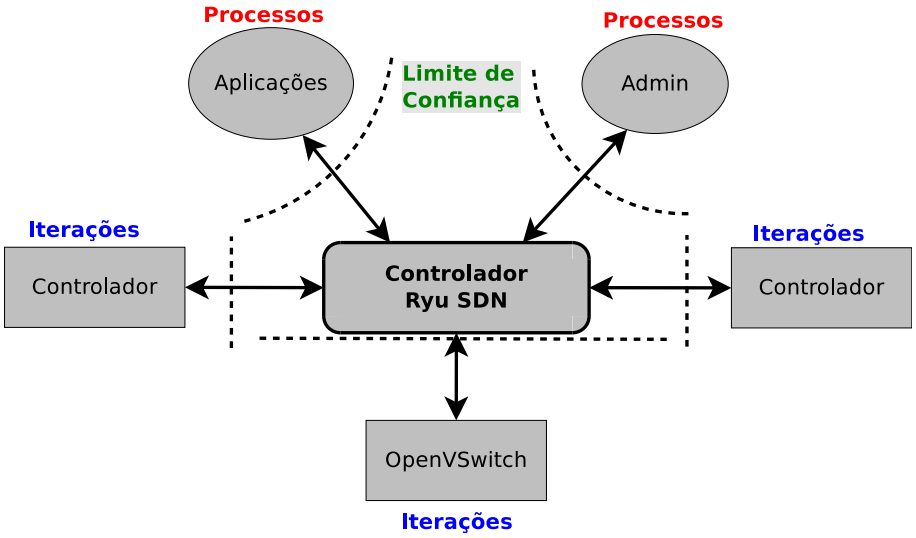


Figura 28 – Fluxo de Dados de um Cotrolador SDN. Adaptado de (ARBETTU et al., 2016)

3.2.4.1 Funcionamento do Sistema Ryu SDN

Switches OpenFlow possuem várias características. Dentre elas, podemos citar algumas funções mais simples, por exemplo:

1. O endereçamento *Media Access Control* (MAC) dos *hosts* conectados no *switch* é retido pelo controlador e armazenado em sua Tabela de Endereços MAC.
2. De acordo com o endereço do pacote que está na Tabela MAC (*learned*), o controlador transfere para a porta onde está conectado o *host* de destino.
3. Quando o endereço dos pacotes não são conhecidos na Tabela MAC, o controlador envia uma mensagem para todo o *data plane* para descobrir o endereço de todos os elementos que estão conectado ao *switch*, *Flooding Message*

A Figura 29 mostra um fluxograma do sistema de *flooding* para mapeamento das portas dos *switches*. O *Packet In Event* será disparado, assim que o primeiro pacote for enviado em direção ao *switch*. Esse evento irá solicitar informação sobre a Tabela de Fluxo por meio de MENSAGENS enviadas ao *data plane*. Assim, se a tabela estiver vazia, ele fará o mapeamento de todo o *data plane*.

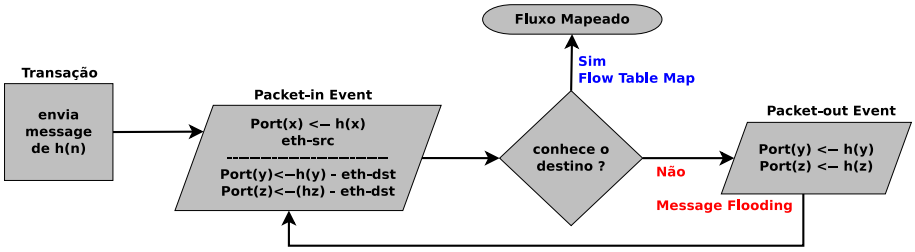


Figura 29 – Fluxograma do Mapeamento de portas (*learning*)

Nesse formato de mapeamento, caso a tabela esteja vazia, um *Packet-Out Event* será disparado enviando todas as informações de *hosts* que estiverem conectados as portas, atualizando a tabela de fluxo. Dessa forma, na próxima requisição, o caminho já estará mapeado no *switch* não necessitando de novo mapeamento. Outra característica, é que pode ser definido um mapeamento temporário, de modo que o fluxo de fonte/destino seja destruído após a transferência de informação, esse recurso foi utilizado no projeto de pesquisa.

Openflow switches executam várias funções que em outros controladores são funções estáticas, podendo executar instruções de acordo com mensagens oriundas dos *switches* via protocolo OpenFlow (*SouthBound*), essas funções são mostradas mais abaixo:

- Reescreve o endereço dos pacotes recebidos ou transfere esses pacotes para uma porta específica.
- Transfere os pacotes recebidos para o controlador (*Packet-in Event*)

- Transfere os pacotes encaminhados para o controlador para uma porta específica (*Packet-Out Event*)

Para finalizar, a Figura 30 apresenta como o controlador com o Ryu SDN preenche sua tabela de fluxo (*Flow Table*), que no momento da inicialização do sistema seu estado é vazio (*empty*).

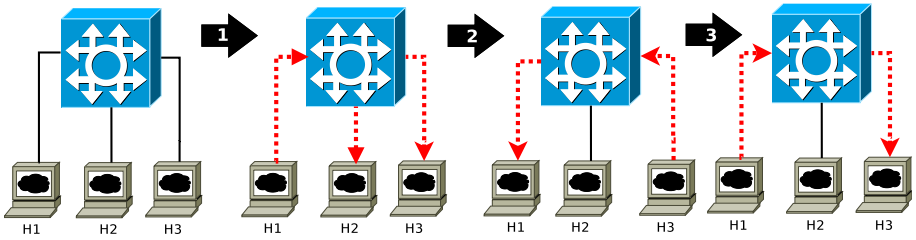


Figura 30 – Mapeamento de Fluxo Ryu SDN.

1. A rede é inicializada, os *hosts* virtuais são configurados e ficam aguardando por novas conexões. H1 envia um pedido de conexão para H3 e o *switch* identifica o *Packet-in Event*, disparando um *Packet-Out Event* com *Action* "OUTPUT: Flooding"para descobrir aonde está conectado o H3.

2. O *switch* recebe a localização da porta de conexão de H3 por meio de um *Packet-in Event* disparando um *Packet-Out Event* com *Action* "OUTPUT: 1", informando para qual porta seus pacotes serão destinados.

3. O controlador faz o mapeamento do caminho de H1 a H3 por meio de um *Packet-in Event* disparando um *Packet-Out Event* com *Action* "OUTPUT: 4".

Tabela 6 – Comparação dos Recursos dos Controladores SDN. Adaptado de (KHONDOKER et al., 2014)

	POX	**Ryu**	**Trema**	**FloodLight**	**OpenDayLight**
Interfaces	SB (OpenFlow)	SB (OF) + OVSDB JSON	SB (OpenFlow)	SB(OpenFlow) + (Java e REST)	SB(OpenFlow) + NBV (Java e RPC)
Virtualização	Mininet e OpenVSwitch	Mininet e OpenVSwitch	Ferramenta Própria	Mininet e OpenVSwitch	Mininet e OpenVSwitch
GUI	Sim	Sim	Não	Web com REST	Sim
REST API	Sim	Sim (apenaspara SB)	No	Sim	Sim
Produtividade	Média	Média	Alta	Média	Média
Open Source	Sim	Sim	Sim	Sim	Sim
Linguagem	Python	Python	Ruby	Java	Java
Plataforma	Linux, Mac e Windows	Linux (maioria)	Somente Linux	Linux, Mac e Windows	Linux
OpenVSwitch	OF v1.0	OF v1.0, v1.2, v1.3	OF v1.0	OF v1.0	OF v1.0
OpenStack	Não Tem	Alta acoplagem	Baixa acoplagem	Média acoplagem	Média acoplagem

A utilização do *framework* Ryu SDN ainda traz outras características importantes, que motivaram para sua escolha no projeto de pesquisa. A Tabela 6 resume os recursos dos mecanismos utilizados pelos controladores, destacando-se: a utilização de interfaces JSON (Ryu), REST (Floodlight) e Java (Floodlight e ODL); a compatibilidade com todas as versões de *openflow* (Ryu); a alta acoplagem com nuvens privadas como Openstack (Ryu). Isso ajudou no direcionamento dos esforços de implementação.

3.2.5 Proxy Reverso e Load Balance

Tecnologias de Load Balance e Web Content são indispensáveis para otimização de requisições para serviços que estão disponibilizados na Internet. Essas tecnologias minimizam problemas de congestionamento e sobrecarga de servidores que são causados pela duplicação do acesso ao mesmo conteúdo disponibilizado o mesmo enlace de comunicação. Logo, essa minimização foi explorada nesta pesquisa com uso de redes SDN para disponibilizar recursos de conteúdo através de redes programáveis, distribuindo os acessos e otimizando o ambiente de rede.

Para tanto, nesta pesquisa foi utilizado o NGINX Load Balance com função de microsserviço, em modo Proxy Reverso e Cache de Conteúdo (CHI et al., 2012). Devido a conexões internas de sistemas operacionais UNIX (*user space*) foi possível a utilização do NGINX com uma instância por domínio, ou seja, duas instâncias em execução ao mesmo tempo, seja no lado LOCAL ou em NUVEM, permitindo que fosse alcançada a escalabilidade da arquitetura.

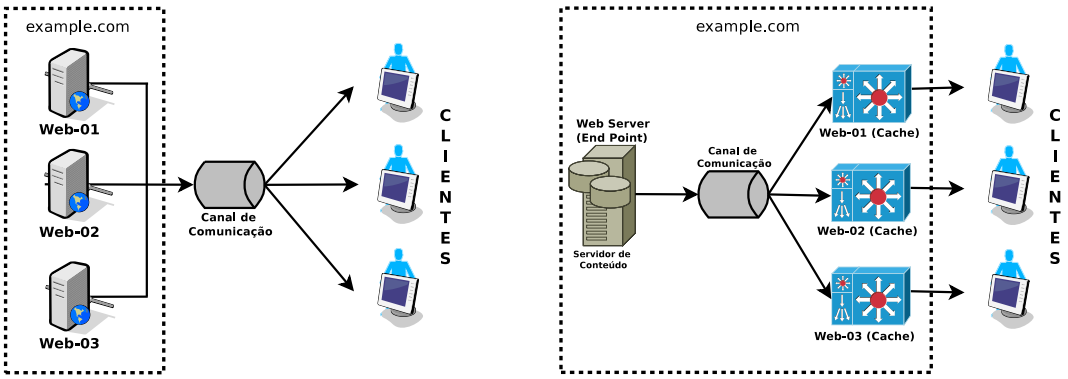
O sistema de proxy do NGINX, foi escolhido por dois motivos: primeiro, a possibilidade de incrementar recursos em um virtual *host*, aumentando o conjunto de serviços disponibilizados em rede, termo conhecido como: (*Scalling Up*); segundo, a possibilidade de conectar esses serviços a outros serviços disponíveis, por exemplo, instâncias de máquinas na nuvem executando outro proxy reverso com os mesmos serviços, termo conhecido como (*Scalling Out*). Essas características motivaram o uso do NGINX (AIVALIOTIS, 2016).

3.2.5.1 Modelo de Distribuição

A replicação do serviço distribuído em um ambiente local e de nuvem, possibilitou a redução da problemática de entrega de conteúdo, por aplicações caras consumindo alta largura de banda de servidores. Nesta pesquisa, Information Centric Networks (ICN) utilizam a rede com desempenho otimizado com cache em aplicações enviando conteúdo pela rede, seja o serviço ativo numa rede LAN ou ativo em Nuvem (*backends*). Portanto, as requisições locais em redes com o melhor posicionamento geográfico não são determinantes para os clientes que fazem requisições ao domínio.

Os servidores estão distribuídos pela rede, encurtando a distância de transmissão de informação e de conteúdo para os clientes. Esses servidores estarão habilitados para efetuar operações de CRUD bem como, uma ampla variedade de conteúdo pela rede. A Figura 31

mostra um exemplo de requisições a arquiteturas convencionais em comparação com requisições executadas para arquiteturas que possuem *backend*.



a) Sistema de Rede Convencional

b) Sistema de Rede de Conteúdo

Figura 31 – Conteúdo distribuído em Redes de Conteúdo.

Na lado (A) da figura, observa-se os serviços distribuídos pelos servidores, entretanto, todo o tráfego origem/destino passa por um único canal de comunicação, o que torna as conexões duplicadas neste canal de comunicação, elevando o consumo da banda aumentando o *overhead* na rede. Esse tipo de cenário motivou o trabalho desta pesquisa. No lado (B) da figura, tem-se o *backend*, ou *front-end*, uma vez que, pequenos sistemas de cache na aplicação web respondem as requisições mais solicitadas, armazenando cópias de conteúdos não modificados em memória para serem posteriormente devolvidos para os requisitantes que os solicitarem. Esses sistemas, ainda contam com um servidor de conteúdo, que para esta pesquisa, foi utilizado como proxy reverso, encurtando o caminho origem/destino, diminuindo o consumo da banda, aumentando desempenho da rede.

3.2.5.2 Sistema de Proxy Reverso

Como dito na introdução da Seção 3.2.5, optou-se por utilizar o sistema NGINX executando em *background*. O sistema intercepta as requisições de clientes para a origem e abre novas conexões para esses servidores, que são denominados de *Upstream*. No meio do caminho, a requisição pode ser dividida de acordo com seu Identificador Uniforme de Recurso (do inglês, *Uniform Resource Identifier* (URI)) (MEALLING; DENENBERG, 2002) por meio de parâmetros pré-definidos direcionando para uma melhor resposta para os clientes, essas características são almejadas nesta pesquisa. Desta forma, alterando qualquer parte do formato original do Localizador de Recurso Uniforme (do inglês, *Uniform Resource Locator* (URL) (MEALLING; DENENBERG, 2002)), possibilitará enviar os pedidos para qualquer caminho através do proxy. Este comportamento também é desejado nesta pesquisa. A Figura 32, apresenta um modelo genérico de repasse de requisições pelo proxy reverso.

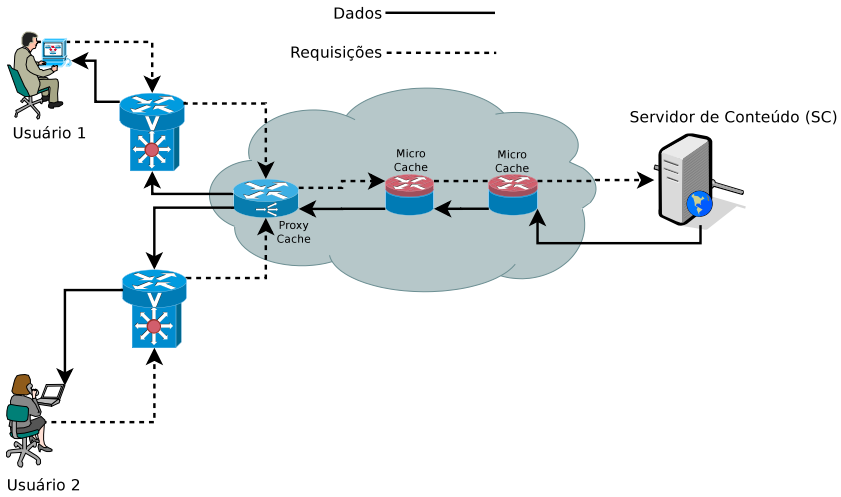


Figura 32 – Proxy Reverso Respondendo Requisições.

As requisições enviadas pelos usuários passam pelo proxy que refaz as requisições ao servidor de conteúdo (Servidor Web) que devolve as requisições para o proxy, e o proxy devolve como forma de resposta aos clientes. A Figura 32 ainda mostra a possibilidade de uso de microcaches web que também poderão responder para o proxy reverso, tornando ainda mais rápido o tempo de resposta.

3.2.5.3 Sistema de Balanceamento de Carga

Sistemas de balanceamento de carga são um dos meios utilizados para tornar altamente disponíveis sistemas considerados como criticos. Logo, para garantir que a nuvem assuma o controle local da arquitetura mantendo os serviços e o tunelamento disponíveis, optou-se por utilizar o KeepAlived para Linux Virtual Server (LVS) (CASSEN, 2002). KeepAlived é um software para roteamento de código aberto escrito em C, que possibilita o controle de servidores virtuais que rodam sob o Kernel Linux. Como a rede SDN utilizada na pesquisa foi executada sob o mesmo kernel linux, significa que o LVS pode ser amplamente utilizado em nível de espaço do usuário, dessa forma, no lado da nuvem, possibilitou utilizar sistemas de *Failover*, com um LVS em Backup, para no caso de falha, o mesmo assumirá o controle até que o LVS Mestre retorne sua operação.

Sistemas LVS podem ser utilizados para construir ambientes altamente escaláveis e altamente disponíveis, característicos dessa pesquisa, por exemplo, web, cache, ftp, e-mail, mídia e Voz sobre IP (VoIP) (KIVITY et al., 2007). LVSs são compostos por:

- **Interface WAN**: uma interface de rede Ethernet que vai ser acessada pelos clientes da rede.
- **Interface LAN**: uma interface que será usada para gerenciar o balanceamento de carga.
- **Linux Kernel**: módulo do kernel utilizado como um roteador.

A alta disponibilidade pode ser alcançada por meio do Virtual Router Redundancy Protocol (VRRP) (HINDEN et al., 2004). Este protocolo elege uma função de roteador virtual, permitindo que o protocolo opere sob várias tecnologias de rede LAN através do suporte *multicast*. Cada instancia VRRP pode definir seu identificador de roteador virtual (VRID) e um conjunto de endereços IP, associando seus endereços reais a uma interface, permitindo o mapeamento de outros roteadores virtuais, realizando o backup dos endereços sem restrição ao uso em LANs diferentes, ou seja, no caso desta pesquisa, o IP do roteador virtual MASTER (LOCAL) é passado para o BACKUP (Nuvem) para que ele assuma toda a rede em caso de falha, mantendo uma cópia fiel do servidor de conteúdo e do proxy reverso na nuvem. Os componentes principais do VRRP utilizado na pesquisa são apresentados na listagem abaixo:

- **Instancia VRRP**: Realiza o backup de uma ou mais instancias VRRP. Nesta pesquisa utilizamos duas instancias VIP1 e VIP2, sendo LOCAL e NUVEM com o mesmo IP.
- **Endereço IP propietário**: a instância VRRP tem apenas um único endereço IP como interface. Essa interface responderá através de conexões TCP ou ICMP para checagem do serviço ativo e backup do IP.
- **Estado Mestre**: responsável por enviar as mensagens para outra instância informando que está on-line.
- **Estado de Backup**: é capaz de assumir o ambiente se o Mestre parar de enviar mensagens.

A Figura 33 apresenta um modelo genérico de instância de um roteador virtual (VRRP) para assumir o encaminhamento de mensagens. Quando o VRRP Master está ativado para o ID 1 com prioridade 100, significa que o mesmo é proprietário do IP Virtual e responsável pelo encaminhamento das mensagens. Quando o VRRP está ativado para o ID 1 com prioridade 101, significa que o Master parou de funcionar, e mesmo sem ser proprietário, assume a responsabilidade pelo encaminhamento das mensagens da LAN local ou da nuvem. Esse recurso será utilizado na pesquisa.

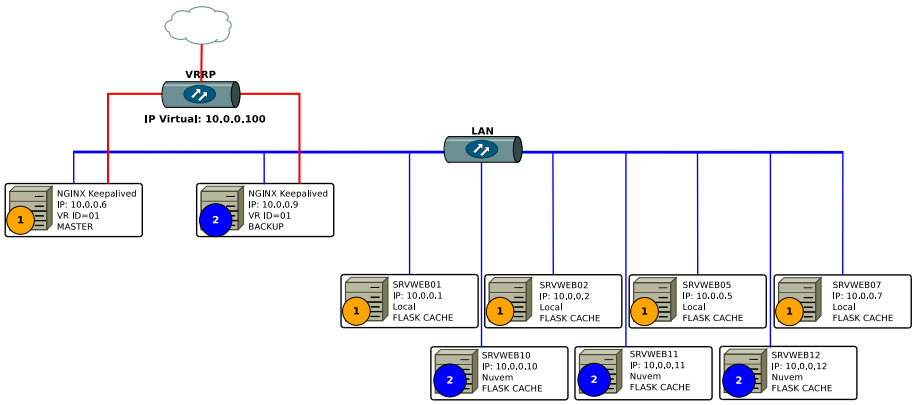


Figura 33 – Balanceamento de Carga com protocolo VRRP.

3.2.6 Sistema WebServer

O Flask é um microframework para Python baseado em Werkzeug e Jinja2 (GRINBERG, 2014), utilizado para desenvolvimento de aplicações para Web utilizando muitos padrões. Este framework foi escolhido para compor a pesquisa devido não ter dependências além das dependências padrão do Python Standard Library (LUNDH, 2001). Este microframework não necessita de suporte de terceiros, como validação de requisições ou bibliotecas para comunicação com bancos de dados. Esses recursos podem ser adicionados com extensões.

O microframework Flask fornece flexibilidade através do seu modelo mínimo para funcionamento, que permite não somente a independabilidade de ser executado paralelamente com outros serviços (LE et al., 2015), mas ter suporte a cache de requisições em memória ou disco (ROCHA, 2014). Esse recurso foi amplamente utilizado na pesquisa. As possibilidades para configuração de caches com Flask posuem uma série de chaves que podem ser configuradas, definindo backend de cache. A chave mais importante para o backend é o CACHE_TYPE, está chave tem configuração nula (null) por padrão, sendo definido o backend para o cache pelo administrador. As opções de backend podem ser descritas logo abaixo:

- **null**: sem Cache – NullCache.
- **simple**: usa pickle (hash) em memória para servidores single process.
- **memcached**: requer um servidor de memcached por meio da biblioeca pylibmc.
- **redis**: Requer redis, werkzeug 0.7 e configurações do Redis no settings.
- **filesystem**: o mesmo que o simple, mais armazenará o pickle em filesystem ao invés de memória. Este recurso será utilizado na pesquisa.
- **gaememcached**: Para o Google AppEngine.
- **saslmemcached**: Mesmo que o memcached.

Este projeto utiliza-se de uma coleção de webservers rodando em Python Flask atuando como um backend para o proxy reverso. Além do script da rede para o Mininet Emulator ser desenvolvido em Python, o Sistema Operacional e os WebServers tambem são escritos em Python, tornando o sistema mais robusto e confiável. Por fim, a Figura 34 mostra um fluxo de uma requisição web quando o sistema possui uma camada como gateway de requisições.

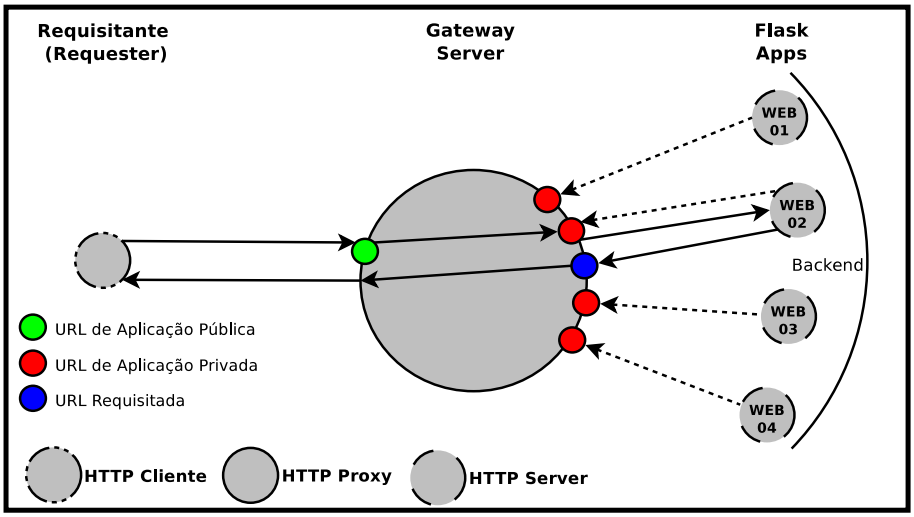


Figura 34 – Fluxo de requisição HTTP com Backend Flask, adaptado de (JONES, 2014).

3.3 Implementação do Sistema de Requisições

O sistema de requisições foi montado com base no paradigma ICN, de modo que as requisições deverão ser aleatórias mesmo se executadas para conteúdos diferentes. Para atender essa premissa, será estabelecido uma camada intermediária antes que a requisição chegue ao servidor de destino, conforme Figura 34. Essa requisição só deverá atingir o servidor de destino se o conteúdo solicitado não estiver presente no servidor de conteúdo (Content Server), ou, caso esteja presente, e o conteúdo não sofreu alterações dentro do tempo de *Timeout*. Se o conteúdo armazenado sofrer alterações dentro do tempo de *Timeout*, esse conteúdo deverá ser novamente requisitado ao servidor de origem para fazer atualização dos arquivos físicos do servidor de conteúdo. Essas operações deverão ser transparentes para os requisitantes do sistema.

A Figura 35 apresenta o fluxo de funcionamento de requisições com otimização dos recursos da banda. Um cliente envia uma requisição de conteúdo (dinâmico para o requisitante) para um domínio previamente definido, que em sua primeira requisição não apresenta cópia (Cache) de conteúdo solicitado, passando a solicitação diretamente para o servidor de origem. O *webserver* faz o roteamento da requisição criando a URI correspondente ao conteúdo solicitado e gera um Hash (Pickle [3] permitindo o uso de cache através de funções simples, como *Decorators* (KONRAD, 2016)) como referência em memória e solicita a cópia do conteúdo em cache (CACHE_TYPE=Filesystem) para gravar fisicamente no disco. Em seguida, o *webserver* retorna os dados originados da cópia em cache e mantém os arquivos inalterados no disco, estabelecendo o Timeout para atualização dos arquivos realizando a entrega dos dados para o requisitante.

De acordo com a continuidade de requisições ao domínio, mostrados na Figura 35, o servidor de conteúdo, checa se há armazenamento em *Filesystem*, então faz uma cópia do cache e assume o controle das respostas para os requisitantes (Reponses), evitando que as requisições atinjam o servidores web no seu endpoint reduzindo o consumo dos recursos. Dado um tempo

[3] ***Pickle*:** é uma forma nativa de serialização que armazena o conteúdo de uma estrutura de dados para que possa ser montado em uma maquina diferente (RHODES et al., 2014)

pré-estabelecido o proxy reverso ficará checando a atualização do cache realizando novas cópias do conteúdo e atendendo as requisições dos clientes do sistema.

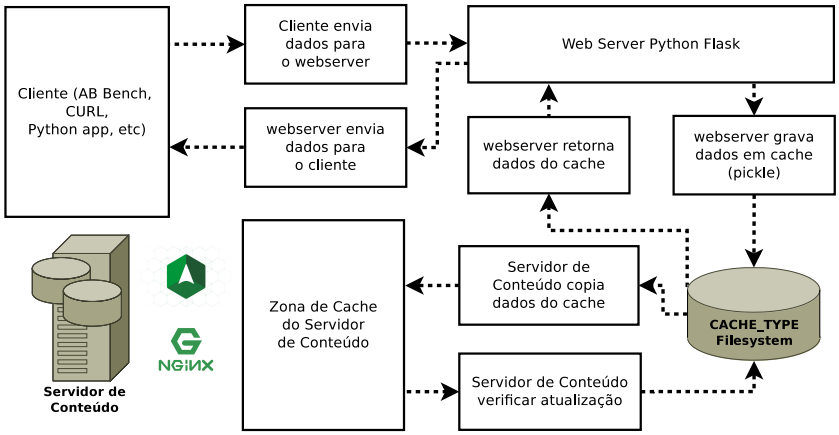


Figura 35 – Fluxo de funcionamento do microframework Flask.

3.4 Métricas do Sistema

O desenvolvimento desta pesquisa, caracteriza-se pela avaliação da otimização dos recursos de uma rede de conteúdo programável, através de um sistema de cache e replicação dos conteúdos distribuídos em um domínio específico, garantindo a disponibilização dos conteúdos no caso de falhas do sistema, onde uma cópia em nuvem, assumirá o controle do plano de dados mantendo a redução do consumo da rede de acordo com a vazão da mesma. Assim, os *workloads* que farão parte dos experimentos devem analisar a eficiência da rede, bem como o ganho obtido com a utilização de SDN/ICN.

3.4.1 Distribuição do Gerenciamento

Considerando que as redes de computadores devem atender a requisitos mínimos de operação, uma vez que, com o crescimento das redes em escala e complexidade, o gerenciamento por um único gerente responsável por todas as funções ainda é viável quando se trata de medição de variáveis do enlace. Para coletar amostras de acordo com a topologia definida, o método de classificação de Leinwand (LEINWAND; CONROY, 1996) e Schönwälder (SCHONWALDER; QUITTEK; KAPPLER, 2000) foi utilizado.

A inicialização de topologia semelhante ao Código 3.1, inclui a utilização da API do Python (*from mininet.topo import Topo*) importando os recursos para criação de topologias customizáveis. O código inclui a definição de Classes para Criação de Topologias (*class MyTopo(Topo):*) bem como, o Método para inicialização e criação (*def init(self):*) da topologia de rede que suportará a arquitetura. A inicialização dos *hosts* e *switches* possibilita a análise de variáveis estaticamente – quando a rede se encontra em repouso aguardando conexões e sem nenhum tráfego, e também a avaliação de dados de variáveis dinamicamente – quando as variáveis sofrem

alterações quando aplicado carga de tráfego na rede. Dessa forma, por exemplo, a vazão total do enlace pode ser alcançada estaticamente tornando-a dinâmica quando executado medições em função do tempo e com diferentes cargas de tráfego aplicadas ao ambiente.

3.4.1.1 Distribuição para Medição de Topologia

Seguindo o paradigma de Leinwand e Conroy (1996), uma **arquitetura centralizada** foi definida para realização de medições. Apenas um gerente e um agente foram responsáveis pelos procedimentos no plano de dados da arquitetura da dissertação (Local e Nuvem), utilizando banco de dados local gerenciado verticalmente.

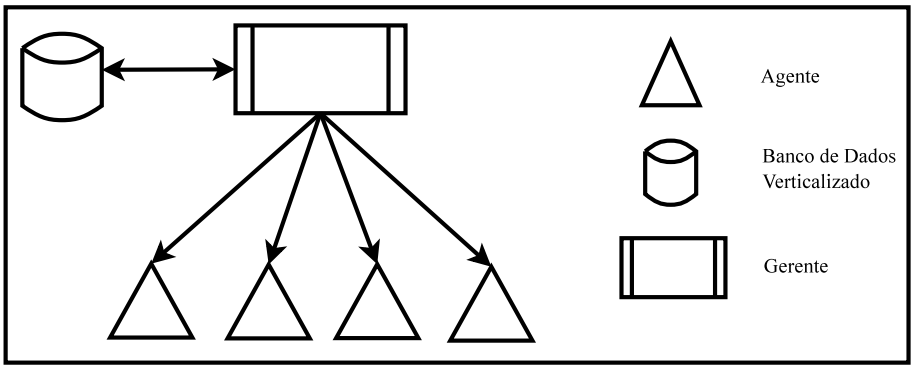


Figura 36 – Modelo Centralizado de Arquitetura de Gerenciamento. Adaptado de (LEINWAND; CONROY, 1996)

As informações foram coletadas através do gerente por conta do único ponto de controle. Não foram contempladas de medições de tolerância a falhas, pois o objetivo foi buscar os valores máximos que podem ser alcançados pelos enlaces no lado Local, Tunelamento e em Nuvem. A rede estará em repouso, ou seja, será realizado medições não intrusivas avaliando o comportamento da rede observando a taxa de chegada de pacotes em um ponto determinado (gerente). Não serão utilizados nenhum dos recursos e/ou serviços da rede mantendo a arquitetura sem nenhuma pertubação. Essa medição pode ser realizada com um sistema cliente-servidor de medições ativas, comparando seus resultados e validando o ambiente com suas capacidades máximas para utilização do tráfego de dados.

Para procedimentos centralizados com o ambiente em produção, a utilização de medições intrusivas foram necessárias sem deixar o modelo centralizado para medições de topologia. Foram injetados pacotes na rede redirecionando a saída de informações para o banco de dados centralizado, executando em seguida a análise os dados. Foram enviadas sequências de pacotes temporizados por meio do *Internet Control Message Protocol* (ICMP) com Pings tipo 0 (*echo request*) obtendo retorno através do Ping tipo 8 (*echo reply*), dessa forma, puderam ser medidos os tempos de *Round Trip Time* (RTT), alcance mínimo e máximo, média e perda de pacotes, validando ou não o ambiente de rede.

Assim, o conjunto de métricas para medição do desempenho da topologia podem ser definidos de acordo com a Tabela 7:

Tabela 7 – Base para Medição de Desempenho da Topologia

RETARDO/PERDA	**AMBIENTE**		
	Local	**Nuvem**	**Local e Nuvem (Tunelamento)**
Retardo de Bidirecional	✓	✓	✓
Retardo Unidirecional	✓	✓	
Retardo de Processamento			
Retardo de Enfileiramento			
Retardo de Transmissão	✓	✓	✓
Retardo de Propagação	✓	✓	✓
Perda de Pacotes Unidirecional			✓
Perda de Pacotes Bidirecional			✓

3.4.1.2 Distribuição para Medição de Requisições

Considerando que a proposta desta dissertação prevê um ambiente distribuído com utilização de múltiplos agentes e gerentes cada um atuando como ponto central dentro de seu alcance de funcionamento, e além das arquiteturas de gerenciamento hierárquico e distribuído presentes no paradigma de Leinwand e Conroy (1996). O paradigma de Schonwalder, Quittek e Kappler (2000) aponta para os sistemas distribuídos com medições intrusivas considerando o envio de requisições HTTP com métodos GET para calcular a largura de banda entre outros. Quatro classes de sistema fazem parte do modelo de Schönwälder, mas para esta pesquisa, apenas dois foram utilizados: ***Gerenciamento Fracamente Distribuído*** quando as medições alcançarem todos os componentes do sistema, isso inclui os *webservices* e o proxy reverso, estando em ambiente local ou em nuvem; e o ***Gerenciamento Fortemente Distribuído*** quando as medições de replicação dos serviços forem utilizadas. Dessa forma, as métricas serão obtidas a partir de um sistema distribuído local e em nuvem, sem passar por pontos centrais.

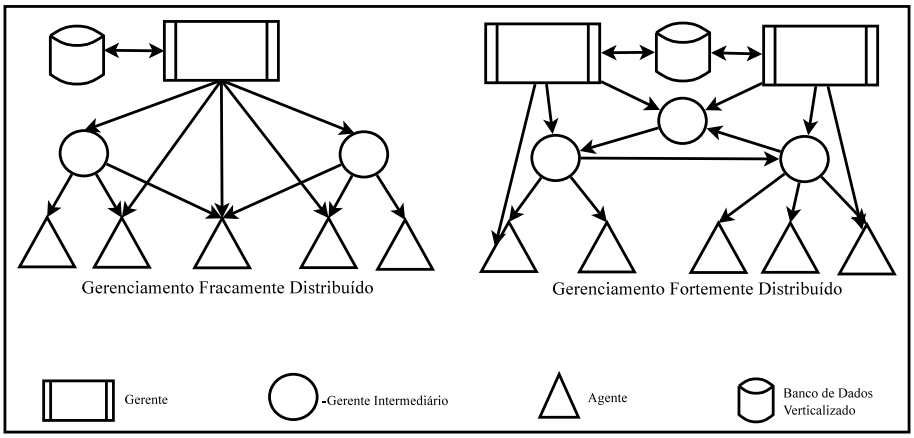


Figura 37 – Modelo Distribuído de Arquitetura de Gerenciamento. Adaptado de (SCHONWALDER; QUITTEK; KAPPLER, 2000)

Em arquiteturas hierárquicas, funções de sistema são divididas por múltiplos gerentes para medição de variáveis dinâmicas distribuídas pelo ambiente, com o banco de dados localizado

no gerente de maior nível, o que contribui para medições do sistema em produção com respostas (*responses*) vindas do cache do proxy reverso para toda a rede. Dois gerentes poderão executar a coleta de informações de medição, mas os dados coletados não precisarão necessariamente ser enviados para o ponto central, poderão ficar armazenados no gerente que demanda a medição (replicação), ou seja, utilização de requisições *Hyper Text Transfer Protocol* (HTTP) (GROUP et al., 1999) através do métodos GET, POST, PUT e DELETE de acordo com parâmetros pré-definidos, replicando o serviço e analisando as possibilidades de ganho de desempenho. Essas requisições serão executadas de acordo com a seguinte formatação:

- Requisições locais diretamente a WebServers para conteúdos específicos.
- Requisições em nuvem diretamente a WebServers para conteúdo específicos.
- Requisições locais para domínio específico e para conteúdos aleatórios (proxiados).
- Requisições em nuvem para domínio específico e para conteúdos aleatórios (proxiados).

Assim, o conjunto de métricas para medição do desempenho do ambiente através de requisições podem ser definidos de acordo com a Tabela 8:

Tabela 8 – Base para Medição de Desempenho do Ambiente

LARGURA DE BANDA	**AMBIENTE**		
	Local	**Nuvem**	**Local e Nuvem (Tunelamento)**
Largura de Banda de Contenção	✓	✓	✓
Largura de Banda Disponível	✓	✓	✓
Largura de Banda Utilizada			
Largura de Banda Alcançável	✓	✓	✓
Cache HIT Ratio	✓	✓	✓
Cache Byte HIT Ratio	✓	✓	✓
Tempo Médio entre Falhas (MTBF)			✓
Tempo Médio entre Reparos (MTTR)			✓

4 Protótipo Programável com Replicação em Nuvem

Este capítulo apresenta as etapas necessárias para desenvolvimento da arquitetura experimental que suporta este trabalho de pesquisa. Será apresentado a arquitetura distribuída em camadas detalhando cada elemento do sistema com suas funções locais e em nuvem. Esses componentes estarão agrupados por camadas e dispostos de maneira a apresentar sua instalação, configuração, funcionamento e relação direta ou indireta com as redes programáveis e com o paradigma de redes de conteúdo.

4.1 Arquitetura de Rede Programável SDN/ICN

Baseado no modelo de interação de APIs de arquiteturas SDN presentes na Figura 5 da Página 33, na arquitetura de roteamento de internet com publicação assinada, do inglês *Publish Subscribe Internet Routing (PSIRP)* da Seção 2.2.2.3 da Página 40, e no contexto genérico de serviços em nuvem e seus elementos, presentes nas Figuras 11 e 12 da Página 43, apresentamos a arquitetura em estudo conforme Figura 38.

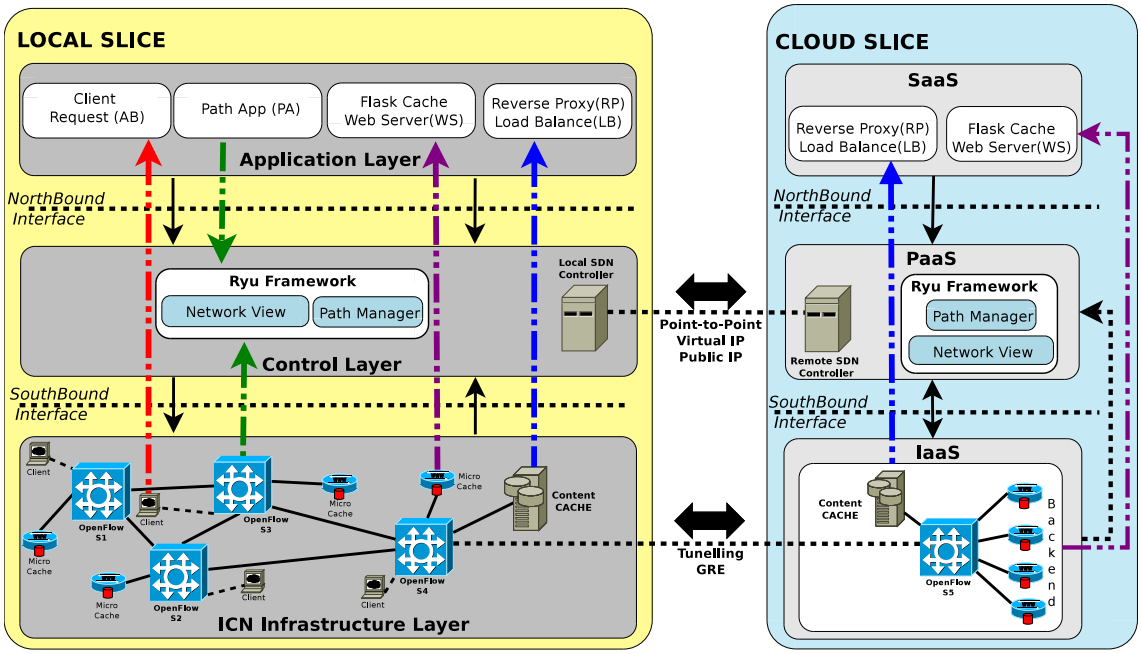


Figura 38 – Arquitetura Programável com Replicação em Private Clouds.

A arquitetura está disposta com duas composições e vários elementos relacionados em cada uma dos lados. As duas Máquinas Virtuais (VM) mostradas na Figura 38 representam uma ICN suportada por SDN, onde de um lado temos a representação do lado ”Local” da rede (*Local Slice*). Do outro lado, temos a representação da replicação do plano de dados em ”Nuvem“

(*Cloud Slice*) com os elementos necessários para o funcionamento do sistema replicado com *backend* para tolerância a falhas.

Esta arquitetura pode funcionar independentemente dos lados estarem conectados através do túnel, funcionando localmente no ambiente em que estiverem iniciados, ou seja, o lado local pode ser inicializado sem que o lado da nuvem esteja em produção e vice-versa. Isso significa que as aplicações, o controlador (orquestrador) e os *switches* continuarão a se comunicar normalmente estando na mesma rede LAN. Logo, o plano de dados local e o controlador local realizarão suas funções normalmente, enviando requisições as camada inferiores, consumindo os serviços sem realização de requisições em direção à nuvem, assegurando a replicação localmente e em memória sem o suporte dinâmico da nuvem para o caso de falhas.

O *Local Slice*, está composto pela maioria dos elementos da arquitetura, visto que o objetivo desta pesquisa é promover um ambiente na nuvem que dê suporte a eventos originados pelo lado local, isso se justifica pela quantidade de aplicações e *switches* que estão presentes no lado local.

O *Cloud Slice*, está composto pelos elementos necessários para que os serviços disponibilizados pelo lado local possam ser suportados na nuvem e atendam os requisitos de funcionamento de uma rede. Assim, justifica-se a não necessidade das mesmas aplicações do lado local estarem na nuvem, uma vez que a nuvem dará suporte para otimização do lado local. Por esse motivo, o lado da nuvem está caracterizado na Figura 38 como serviços distribuídos de computação na nuvem.

Por fim, nas próximas Seções, os elementos desta arquitetura serão detalhados em sua totalidade.

4.1.1 Camada de Aplicação (*Application Layer*)

Na Camada de Aplicação (*Application Layer*) estão localizados às aplicações em nível de usuário que compõem o *Local Slice* e o *Cloud Slice*. Esta camada está localizada no topo da arquitetura e se comunica com o plano de dados e com o controlador através da interface *northbound*. Essas requisições em nível de usuário são executadas pelo Apache Benchmarking (AB) (RAHMEL, 2013) dito por vários autores como a melhor ferramenta para aplicação de testes de requisições web. Para atender estas requisições, também está nesta camada uma aplicação com microframework Flask WebServer (WS) que será instanciada pelo controlador e será executada em vários pontos da rede seja local ou nuvem. Sendo assim, esse conjunto de aplicações em nuvem, podem ser caraterizam-se como SaaS, de acordo com Seção 2.3.2.1 da Página 45

Para gerenciamento da rede, um conjunto de ferramentas do próprio sistema operacional da rede é disponibilizado, o que chamamos de Path App (PA), conforme Seção 3.2.4 da Página 74. Através de comandos HTTP é possível fazer alterações em fluxos, definir origens e destinos, coletar informações de *switches*, definir rotas, entre outros. Essa caixa de ferramentas se comunica diretamente com o S.O da rede por meio da *northbound interface*.

Por último, está a aplicação para proxy reverso (*Reverse Proxy (RP)*) e balanceamento de carga (*Load Balance (LB)*) conforme Seção 3.2.5 da Página 78. Trata-se de duas aplicações distintas interoperáveis, ou seja, que podem se comunicar utilizando-se dos seus recursos. O proxy reverso atuará na função de repasse de requisições assim que executar o cache do conteúdo mais solicitado, liberando o tráfego para os servidores web que são *backends* no lado local. O LB local entregará as requisições web ao LB da nuvem, caso o proxy apresente falhas, fazendo com que o lado da nuvem assuma o *backend* local, de acordo com a Seção 3.2.5.3 da Página 80.

4.1.2 Camada de Controle (*Control Layer*)

Na Camada de Controle *Control Layer* está localizado o S.O do controlador. Nesta pesquisa utilizou-se dois principais recursos deste S.O, a visualização da rede (*Network view*) e o gerenciamento da topologia (*Path Manager*). A nível de virtualização e comunicação entre os hospedeiros da rede programável estabelecemos uma conexão ponto-a-ponto para que pudéssemos interligar os controladores remotamente, sendo um controlador no lado local, e o outro localizado na nuvem. Essa configuração serve para endereçamento de IPs públicos que encapsulam o tráfego do tunelamento que foi configurado em um dos *switches* programáveis, caracterizando o Cloud Slice como uma Plataforma como Serviço (PaaS) de acordo com a Seção 2.3.2.2 da Página 45

Os controladores local e nuvem conectados, cada um com seu S.O respectivo (RyuOS), serão responsáveis pela criação e destruição de fluxos no caminho da rede. O controlador obtém informações sobre a capacidade e requisições exigidas dos elementos da rede através do tráfego presente nos fluxos estabelecidos, implementando políticas para as requisições originadas na Camada de Aplicação, e políticas impostas para o encaminhamento de pacotes para a Camada de Infraestrutura (SINGH; JHA, 2017). Dessa forma, o controlador sincroniza o status da rede para aplicar as funções de tomada de decisão. Logo, nesta pesquisa, a visualização e o monitoramento da rede permitiram a utilização das seguintes funções enumeradas logo abaixo:

1. Utilização de linguagem de alto nível (Python nesta pesquisa)
2. Utilização de processo automatizado de atualização de regras (Updates automáticos)
3. Um processo de comunicação no espaço do usuário para coleta de informações da rede.
4. Um processo de sincronização do status da rede.

4.1.3 Camada de Infraestrutura (*ICN Infrastructure Layer*)

Na Camada de Infraestrutura *Infrastructure Layer* estão localizados os elementos que compõem os ativos da rede, ou seja, o plano de dados de uma rede de computadores (*Data Plane*). Neste contexto, *Switches Ethernet*, *Switches Ópticos*, Roteadores, *Switches Virtuais* poderão ser utilizados para formar o plano de dados. Porém, para esta pesquisa, houve necessidade apenas

da utilização de *Switches Virtuais* e Roteadores compatíveis com tecnologias de tunelamento.

Os comutadores de pacotes presentes no ambiente podem ser interligados por meio de diferentes enlaces. Entretanto, como a pesquisa busca ambientes de rede corporativos ou de grande escala, optou-se pela utilização de enlaces cabeados (*wired*) e tunelados. A replicação na nuvem possui *backend* próprio, ou seja, um conjunto de webservers que assumem a entrega de requisições através de um fluxo de dados configurados na própria nuvem, caracteriza o plano de dados da nuvem como uma Infraestrutura como Serviço (IaaS), conforme Seção 2.3.2.3 da Página 46.

4.2 Arquitetura Experimental

De acordo com a arquitetura apresentada na Seção anterior, o ambiente experimental pode ser implementado e testado. A estrutura para realização dos experimentos pode ser representada pela Figura 39. Esta pesquisa não objetiva a substituição de técnicas de integração de SDN/ICN, mas busca disponibilizar um complemento por meio de *middlebox* para adaptação de ICNs utilizando SDN de uma forma simplificada, otimizando a largura de banda degradada pela duplicação da transferência de conteúdo pelo mesmo segmento de rede com replicação em nuvem.

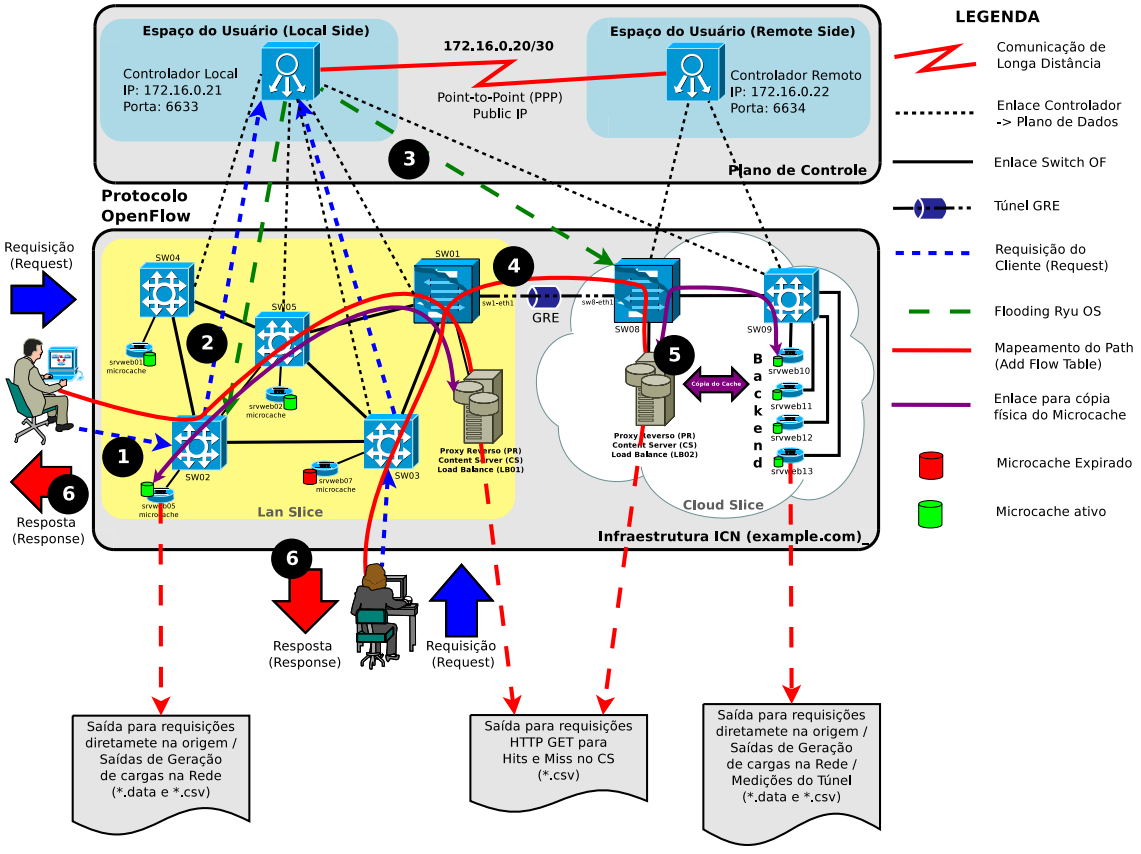


Figura 39 – Funcionamento do protótipo no *request* inicial.

O protótipo SDN para replicação de conteúdo ICN na nuvem foi desenvolvido utilizando

duas máquinas virtuais com o ambiente de emulação Mininet 2.2.2, disponibilizada em (MININET, 2014). Esses ambientes de emulação estão instalados em um Sistema Operacional Ubuntu 14.04.5 LTS, codinome Trusty Tahr. Nas duas VMs, foram criadas topologias específicas para cada lado (*slice*) da arquitetura, sendo que, no lado local, utilizamos uma topologia em árvore conforme descrito na Figura 22 da Página 69, e no lado da nuvem, adotamos uma topologia Linear, conforme Figura 21 da Página 68, uma vez que a infraestrutura da nuvem é menor que a local. As duas topologias utilizadas funcionam com o controlador RyuOS, disponibilizado em (RYU, 2017) e *switches* do tipo Open vSwitch, dispostos da seguinte maneira: 1) 5 (cinco) *switches* do tipo Open vSwitch estão configurados para o lado local da arquitetura; 2) 2 (dois) *switches* do tipo Open vSwitch estão configurados para o lado da nuvem, ambos podem se comunicar através de tunelamento com IP flutuante comunicando o plano de dados com o controlador através do protocolo OpenFlow (OF).

A linguagem utilizada para desenvolver toda a arquitetura foi exclusivamente Python. Devido sua facilidade de programação e ampla compatibilidade com desenvolvimento de aplicações para redes de computadores. Além disso, o controlador RyuOS é totalmente desenvolvido em Python e totalmente gratuito, motivando o desenvolvimento dos *WebServers*, e da própria topologia do sistema.

Para as medições, *scripts Shell* foram desenvolvidos para coleta dos dados e geração de carga na rede. Um *host* aleatório localizado no espaço geográfico do ambiente executará uma aplicação cliente/servidor para podermos coletar os dados do enlace da rede de acordo com as métricas definidas na Seção 3.4. Por exemplo, um *host* aplica uma carga na rede como cliente e o servidor responde ao cliente com parâmetros para a carga aplicada escrevendo sua saída em arquivos ".data". Em seguida os dados são tratados disponibilizando o conjunto para as métricas. Capturando dados dessa maneira, nos permitiu a análise de cada pacote enviado pelo cliente obtendo dados de **retardo** e **perda de pacotes**. Esses dados serão obtidos através do envio mensagens de eco *Internet Message Control Protocol (ICMP)* (POSTEL et al., 1981). Os pacotes serão enviados durante um tempo pré-estabelecido, analisando o ambiente local e depois seu enlace em direção da nuvem obtendo dados sobre o túnel.

Para dados de requisições aleatórias, os próprios *webservers* servirão para geração dos dados. O sistema de cacheamento do proxy reverso foi desativado para obtermos métricas de requisições direto na origem, ou seja, direto nos *webservers* passando pelo proxy. Para realização deste experimento, dois *hosts* fizeram as requisições redirecionando sua saída para arquivos "*.csv". Através de requisições *web* foi possível realizar medições distribuídas de acordo com as métricas definidas na Seção 3.4, por exemplo, **largura de banda de contenção**, **largura de banda alcançável**, **caches hits e miss** entre outros. Com o cache (Content Cache) ativo no proxy reverso, poderemos medir os ganhos da rede pela utilização desse recurso obtendo os dados a partir de *scripts* e aplicações específicas, onde o administrador deverá informar os endereços e IPs dos servidores e as portas para que aconteça o estabelecimento de conexões.

Para tratamento das informações necessárias, foram considerados o uso de ferramentas

gratuitas e desenvolvimento de conjunto de *scripts* próprios com base nos parâmetros dessas ferramentas. Assim o conjunto de aplicações para geração de carga no protótipo é apresentado:

- Um **injetor** de pacotes na rede que funciona com um servidor parametrizado que ficará ”ouvindo” requisições, e um cliente que ficará responsável pela geração de carga na rede de acordo com parâmetros definidos para execução.
- Um enviador de **Eco** ICMP para obtenção de retardo *Round-Trip* na comunicação com os *hosts* e Perda de Pacotes.
- Dois **geradores** de GET HTTP que funcionaram enviando mensagens direcionadas ou aleatórias, por tempo determinado sequencialmente, ou por quantidade de requisições sequencialmente para obtenção dos tempos de resposta de requisições HTTP.
- Um ***Script*** com *loop* infinito que funciona enviando GET HTTP por segundo obtendo o tempo de resposta de falha e de reparos.

4.2.1 Etapas da inicialização da Requisição Inicial

Conforme a Figura 39, as etapas iniciais que ocorrem no sistema quando da ocorrência da requisição inicial, acontecem da seguinte forma:

1. Clientes enviam requisições (*requests*) ao domínio http://example.com conectados ao enlace dos *switches* SW02 e SW03.
2. SW02 e SW03 recebem a requisição e enviam o pedido de mapeamento para o controlador.
3. O controlador verificar a tabela de fluxo (if id_path==null) para enviar um *flooding* na rede para descoberta dos elementos conectados aos *switches*.
4. O controlador executa o mapeamento de origem/destino e adiciona (id_path) na tabela de fluxo (*flow table*) após SW02 e SW03 anunciarem as conexões.
5. O proxy reverso intercepta a requisição e faz o cache físico do conteúdo requisitado, passando a responder as requisições dos clientes liberando o tráfego para os *endpoints* SRVWEB05 e SRVWEB07.
6. Clientes recebem resposta (*responses*).

4.3 Implementação da Topologia

O primeiro passo para inicialização do protótipo é a inicialização da topologia. Essa inicialização permitirá a criação de uma topologia customizada desenvolvida exclusivamente para

atender aos objetivos da pesquisa. Neste trabalho não foram explorados medições e comparações com trabalhos de outros autores a respeito das alterações ocorridas na transmissão devido à mudança de topologia. Isso descaracterizaria a pesquisa e poderia nos direcionar para uma outra abordagem. Este protótipo busca a avaliação de uma arquitetura com vistas a uso de serviços em nuvem.

O emulador Mininet permite a utilização de topologias customizáveis de duas maneiras, como descrito na Seção 3.2.2.3 da Página 69. A primeira, criando o objeto **topo** importando a **Classe Topo** e executando o *script* como módulo através do comando:

$: *mn –custom ~/pastaDoCode/nomeDoArquivo.py –topo mytopo*

A segunda, executando diretamente como programa em um terminal Shell, facilitando o desenvolvimento da topologia. Nesse caso, a topologia é criada sob condições especiais quando uma única função "main()" é encarregada de realizar a chamada para execução de todo o sistema. Logo, nesta proposta todos os *scripts* foram desenvolvidos para serem executados como programa com a chamada da execução direta no Shell, conforme descrito abaixo:

$: *./~/pastaDoCode/nomeDoArquivo.py*

4.3.1 Topologia da Nuvem (Linear)

A rede que será virtualizada para compor o *backend* da proposta será com uma topologia Linear, conforme Figura 21 da Página 68. Nas topologias lineares todos os *switches* são conectadas com o controlador (**Controlador C0**) principal de sua LAN, em seguida esses comutadores são conectados diretamente uns aos outros através de *links* ponto a ponto. A ordem de inicialização ocorre com o lado da nuvem construído antes do lado local, uma vez que a nuvem deverá ficar aguardando por conexões remotas em sua interface externa no *User Space*.

A partir desse instante, são importados todas as classes contendo os objetos necessários para criação dos elementos da rede. A importação dessas classes segue a sintaxe: "***from mininet.NomeDoObjeto import NomeDaClasse***". Feita a importação dos objetos das classes o método principal faz uma chamada numa função interna ("*def cloudtopo()*") e a rede é criada conectando todos os seus elementos. O Algoritmo 32 escrito em pseudocódigo, demonstra as etapas de construção da topologia linear que é utilizada nesta pesquisa.

Algoritmo 1: PSEUDO CÓDIGO DA INICIALIZAÇÃO DA TOPOLOGIA LINEAR

```
   Entrada: host H; controlador C0; switches S;
   Saída: Plano de Dados em Nuvem Inicializado
 1 início
 2     Importar Objetos de Classes;
 3     criaTopo();
 4         ipC0Local ← "IP Público"                    ▷ Bloco que define o controlador;
 5         ipC0Nuvem ← "IP Localhost";
 6         cria o objeto net ← vazio;
 7         cria o controlador c0 ← adicionando (nome, tipo, porta);
 8         cria switch s8 ← adicionando (nome, tipo);
 9         cria switch s9 ← adicionando (nome, tipo);
10     para i ← 9 até 13 faça
11         cria host hi ← adicionando (nome, ip , porta)        ▷ Laço que define hosts;
12     fim
13         cria objeto s8s9 com parametros do link (banda, delay, perda);
14         net conecta (s8, s9, tipo do link, s8s9);
15     para i ← 9 até 13 faça
16         net conecta (hi, s9)                         ▷ Laço que conecta os hosts;
17     fim
18     para i ← 9 até 13 faça
19         hi (configura hi-eth0 com mtu 1400)          ▷ Laço que define o segmento max. do enlace;
20     fim
21         h9 inicializa o Balanceamento de Carga;
22         h9 inicializa o Proxy Reverso;
23         c0 → inicializar();
24         net conecta s8 → c0;
25         net conecta s9 → c0;
26         s8 cria porta para túnel GRE (s8-gre1 ← ipC0Local)   ▷ Estabelece túnel entre s8 e s6 ;
27         Terminal(net);
28     se nomoDoProg == "chamadaPrincipal" então
29         ativaLogs();
30         criaTopo()                    ▷ Chama a função principal e inicializa (main);
31     fim
32 fim
```

De acordo com a sequência acima a Figura 40 mostra o modelo lógico da topologia, que ao final de sua execução, a rede e seus ativos estarão criados e conectados, conforme código disponível no Apêndice A.2.

4.3.2 Topologia Local (Árvore)

Dando continuidade aos passos iniciais para inicialização da arquitetura, a topologia em árvore é apresentada de acordo com o Algoritmo 45. Os blocos principais do pseudocódigo estão comentados, e o código fonte está disponível no Apêndice A.1.

Algoritmo 2: PSEUDO CÓDIGO DA INICIALIZAÇÃO DA TOPOLOGIA EM ÁRVORE

```
   Entrada: host H; controlador C0; switches S;
   Saída: Plano de Dados Local Inicializado
 1 início
 2     Importar Objetos de Classes;
 3     criaTopo();
 4         ipC0Nuvem ← "IP Público"                    ▷ Bloco que define o controlador;
 5         ipC0Local ← "IP Estático";
 6         cria o objeto net ← vazio;
 7         cria o controlador c0 ← adicionando (nome, tipo, porta);
 8     para i ← 1 até 5 faça
 9         cria switch si ← adicionando (nome, tipo)          ▷ Laço que define switches;
10     fim
11     para i ← 1 até 8 faça
12         cria host hi ← adicionando (nome, ip , porta)      ▷ Laço que define os hosts;
13     fim
14         cria objeto s1s2 com parâmetros do link (banda, delay, perda)   ▷ Bloco que conecta os switchs ;
15         net conecta (s1, s2, tipo do link, s1s2);
16         cria objeto s1s3 com parametros do link (banda, delay, perda);
17         net conecta (s1, s3, tipo do link, s1s3)
18         cria objeto s2s4 com parametros do link (banda, delay, perda);
19         net conecta (s2, s4, tipo do link, s2s4)
20         cria objeto s3s5 com parametros do link (banda, delay, perda);
21         net conecta (s3, s5, tipo do link, s3s5)
22         net conecta (h1, s2)                        ▷ Bloco que conecta os hosts;
23         net conecta (h2, s2);
24         net conecta (h3, s5);
25         net conecta (h4, s5);
26         net conecta (h5, s3);
27         net conecta (h6, s1);
28         net conecta (h7, s3);
29         net conecta (h8, s4);
30     para i ← 1 até 8 faça
31         hi (configura hi-eth0 com mtu 1400)        ▷ Laço que define o segmento max. do enlace;
32     fim
33         h6 inicializa o Balanceamento de Carga;
34         h6 inicializa o Proxy Reverso;
35         c0 → inicializar();
36     para i ← 1 até 5 faça
37         net conecta si → c0                        ▷ Laço que conecta os switches em c0;
38     fim
39         s6 cria porta para túnel GRE (s6-gre1 ← ipC0nuvem)      ▷ Estabelece túnel entre s6 e s8 ;
40         Terminal(net);
41     se nomoDoProg == "chamadaPrincipal" então
42         ativaLogs();
43         criaTopo()                          ▷ Chama a função principal e inicializa (main);
44     fim
45 fim
```

A rede que será virtualizada no lado intitulado como **lado local** utiliza uma topologia em árvore (*tree*), conforme descrição da Figura 22 da Página 69. Assim, da mesma forma que as lineares, o controlador c0 envia e recebe informações de todos os *swtiches*, porém esses concentradores estarão dispostos hierarquicamente com os caminhos perfeitamente casados, ou seja, foi projetada uma rede de árvore balanceada para não haver diferenças de velocidades de propagação e do tráfego de uma extremidade para a outra. A motivação para uso em árvore se

aplica para o caso da replicação de requisições para origens aleatórias onde o tráfego deverá fluir sob diferentes níveis mas com o mesmo desempenho, seja para requisições locais ou diretamente para o *backend* da nuvem. O Algoritmo 45 mostra a sequência para criação da rede do lado local. O código desta topologia está disponível no Apêndice A.1.

De acordo com os algoritmos apresentados seções anteriores, a Figura 40 mostra o modelo lógico da topologia, que ao final de sua execução, a rede e seus ativos estarão criados e conectados conforme código disponível do Apêndice A.2.

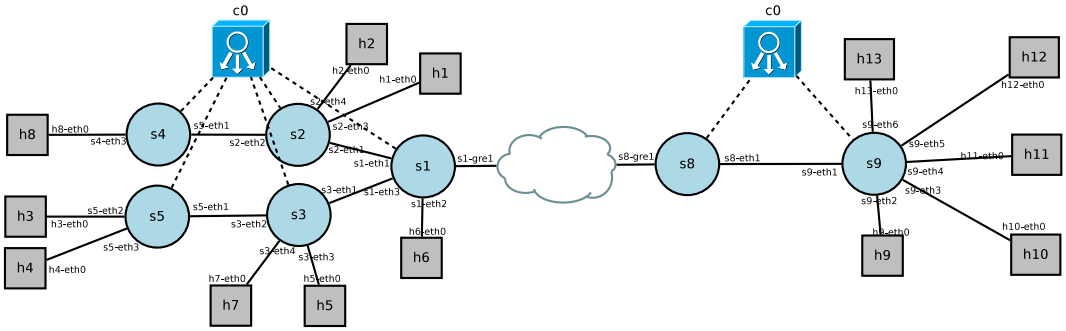


Figura 40 – Topologia Geral do Protótipo da Pesquisa.

4.4 Implementação do Sistema Operacional de Rede

Nesta Seção serão explanados os componentes internos do sistema operacional da rede. Na Seção 3.2.4 da Página 74, foi contextualizado o Ryu SDN Framework em linhas gerais e de funcionamento. Neste ponto, serão apresentados as classes e métodos utilizados para composição da aprendizagem (*learning switch*) da rede por *switches* OpenFlow com Ryu OS.

Ryu OS é uma estrutura SDN baseada em componentes de *software*. Esse *framework* contém uma API bem definida que permite a criação de novos sistemas de controle e gerenciamento de redes SDN. Ryu também é compatível com vários protocolos de gerenciamento de redes SDN tais como: OpenFlow (todas as versões), Netconf, OF-Conf (RYU, 2017).

Para esta pesquisa, foram implementados recursos de controle de pacotes e tráfego utilizando *software* de terceiros. Assim, foi necessário o uso das APIs: Classe para inicialização e definição, Classe para controle dos pacotes e da tabela de fluxo. Essas classes e outras funcionalidades podem ser obtidas por meio das bibliotecas conforme a Tabela 9.

Tabela 9 – Módulos necessários para construção de sistemas com Ryu Framework

Módulo	Componente	Descrição
base	app_manager	Carrega a aplicação e encaminha mensagens para outras aplicações
controller	ofp_event	Definições para eventos do OF
controller.handler	MAIN_DISPATCHER	Define o switch como L2/L3
controller.handler	CONFIG_DISPATCHER	Permite os switch recebam mensagens de configuração
controller.handler	set_ev_cls	Permite gerenciar as mensagens dos switches
ofproto	ofproto_versão	Importa as funções de acordo com as versões.
lib	packet	Instância para códificar e decodificar os pacotes
lib	ethernet	Instância para códificar o endereço MAC como String.

4.4.1 Classe de Inicialização

Dada a etapa de inicialização da rede concluída, foi feita a carga do sistema operacional no ambiente. A inicialização do S.O que fará a orquestração do plano de dados possui uma classe para definição e inicialização. Essa classe é herdada do "***ryu.base.app_manager***" e irá conter uma variável constante para receber a versão de operação do protocolo OF, e dentro do método inicial um dicionário para armazenar os endereços MAC dos elementos da rede associados em um *openvswitch*. O Código 4.1 mostra o exemplo dessa inicialização.

```
class OpenVSwitch13(app_manager.RyuApp):
        OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

        def __init__(self, *args, **kwargs):
                super(OpenVSwitch13, self).\
                __init__(*args, **kwargs)
                self.mac_to_port = {}
# ...
```

Código 4.1 – Classe de Inicialização

A constante OFP_VERSIONS recebe a versão do protocolo OF, e o método "***def __init__***", possui o construtor do OpenVSwitch e o dicionário para os endereços MAC em "***mac_to_port =*** ". Com essa declaração somados a importação dos objetos das classes o sistema operacional da rede poderá iniciar o controle.

4.4.2 Evento Controlador

Depois da inicialização da classe de definição, o controlador necessita habilitar os *switches* para envio e recebimento de mensagens vindas do controlador e vice-versa. Para isso, um evento para tratamento de mensagens é implementado com um evento controlador (*event handler*) correspondente aos tipos de mensagens e comportamentos que os elementos concentradores do

plano de dados devem executar quando as mensagens forem enviadas ou recebidas (RYU, 2016).

O componente ***set_ev_cls*** define o argumento para a função que define o tipo de mensagem e status do OpenVSwitch. Por exemplo, para as mensagens de *Packet-In*, que são as mensagens que chegam no *switch*, o controlador recebe a mensagem aplicando o *event handler* de acordo configuração feita no argumento do ***set_ev_cls*** conforme declaração: "***@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)***".

A partir da execução dessa declaração, um *Three-way Handshake* (3WHS) é estabelecido entre o controlador e o *switch* criando a tabela de fluxo para armazenar as ligações de origem e destino dos elementos do plano de dados. A Tabela 10 mostra as configurações para recebimento de mensagens dos elementos concentradores da rede.

Tabela 10 – Mensagens de Controle para Switches OF com Ryu

Definição (ryu.controller.handler)	**Funcionamento**
HANDSHAKE DISPATCHER	Troca Mensagem HELLO para estabelecer o 3WHS
CONFIG DISPATCHER	Aguarda o recebimento de mensagens do switch
MAIN DISPATCHER	Define o status do switch como normal
DEAD DISPATCHER	Desconecta a conexão

A tabela de fluxo criada para *switches* virtuais inicia com prioridade 0 (zero), ou seja, qualquer pacote que chega ao elemento de rede é prioritário. Essa ação de criação da tabela de fluxo vazia, gera uma ação para criação de uma instância de saída **OUTPUT:port** transferida para a porta do controlador. O controlador verifica qual é a porta de saída e caso o valor seja vazio (id_path == null), o controlador prepara o envio dos pacotes para todas as portas, definindo sua prioridade para 0 (zero) executando o método ***addFlow()*** para enviar um ***Flow Mod Message*** mapeando a origem e o destino dos pacotes. Esse mapeamento ocorre para o tráfego de pacotes dentro do *switch* (comutação de pacotes) e da porta que recebe o pacote e para onde vai no destino final, realizando o mapeamento reverso para o retorno do pacote.

4.4.3 Montagem da Tabela de Fluxo (*Flooding*)

A atualização do *path* por meio do endereço MAC é obtido com a chegada do pacote no *openvswtich*, possibilitando que o MAC de destino e o MAC de origem possam ser obtidos do cabeçalho *ethernet* do pacote. Dessa forma, o *switch* aprende qual a porta que está originando a requisição e verifica em sua tabela, caso o endereço MAC de destino ainda não esteja armazenado, prepara para o envio do *flood* para descoberta da localização do MAC de destino, e em qual porta do *openvswitch* o destino pode ser encontrado, realizando a atualização da tabela. O Código 4.2 mostra o método para este evento.

```
def _packet_in_handler(self, ev):
# ...
        in_port = msg.match["in_port"]
        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]
        dst = eth.dst
        src = eth.src
        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})
        self.logger.info("packet in %s %s %s %s", \
        dpid, src, dst, in_port)

# Aprende o endereco MAC para enviar\
  um flood para toda a rede
        self.mac_to_port[dpid][src] = in_port
        if dst in self.mac_to_port[dpid]:
                out_port = self.mac_to_port[dpid][dst]
        else:
                out_port = ofproto.OFPP_FLOOD
        actions = [parser.OFPActionOutput(out_port)]

# Instala o fluxo de origem e destino \
  evitando um novo flood na rede
        if out_port != ofproto.OFPP_FLOOD:
                match = parser.OFPMatch(in_port=in_port, \
                eth_dst=dst)
                self.add_flow(datapath, 1, match, actions)
# ...
```

Código 4.2 – Método de Controle do *Packet-In*

A saída lógica para todas as portas de todos os concentradores do plano de dados dessa pesquisa, foram configurados para NORMAL por meio da declaração:

"***@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)***"

Nesse evento de controle, os concentradores passam a operar com funções de comutação de pacotes de camada L2 e L3 atendendo os objetivos da pesquisa. Ao final da execução inicial, os elementos da rede foram mapeados de acordo com a Tabela 11.

Tabela 11 – Mapeamento dos OpenVSwitch pelo RyuOS

TABELA DE S1							
Requisições				**Mapeamento Interno**		**Mapeamento para h(n)**	
Orig. IP	Orig. Port	dstIP	dstPort	Packet-in	OUTPUT:port	Packet-in	OUTPUT:port
10.8	52236	10.6	80	01	02	02	S1-GRE1
10.6	33064	10.13	80	02	S1-GRE1	01	02
10.13	80	10.6	33064	S1-GRE1	02	02	01
TABELA DE S2							
Orig. IP	Orig. Port	dstIP	dstPort	Packet-in	OUTPUT:port	Packet-in	OUTPUT:port
10.8	59232	10.1	80	02	03	03	02
10.8	53010	10.2	80	02	04	04	02
TABELA DE S3							
Orig. IP	Orig. Port	dstIP	dstPort	Packet-in	OUTPUT:port	Packet-in	OUTPUT:port
10.3	44112	10.5	80	02	03	03	02
10.3	34442	10.7	80	02	04	04	02
TABELA DE S4							
Orig. IP	Orig. Port	dstIP	dstPort	Packet-in	OUTPUT:port	Packet-in	OUTPUT:port
10.8	59320	10.1	80	01	03	03	01
TABELA DE S5							
Orig. IP	Orig. Port	dstIP	dstPort	Packet-in	OUTPUT:port	Packet-in	OUTPUT:port
10.3	49740	10.7	80	01	02	02	01
10.4	56788	10.5	80	01	03	03	01
TABELA DE S8							
Orig. IP	Orig. Port	dstIP	dstPort	Packet-in	OUTPUT:port	Packet-in	OUTPUT:port
10.8	33028	10.10	80	S8-GRE1	01	01	02
10.10	80	10.8	33028	02	S8-GRE1	02	01
TABELA DE S9							
Orig. IP	Orig. Port	dstIP	dstPort	Packet-in	OUTPUT:port	Packet-in	OUTPUT:port
10.8	33028	10.10	80	01	03	03	01
10.10	80	10.8	33028	03	01	01	03

Por fim, pode ser observado na Tabela 11, nos *switches* S1 e S8 o mapeamento interno para o túnel GRE. Isso ocorre porque são nesses *switches* que se encontram os proxies reversos. Quando a requisição é executada no domínio **http://example.com** configurado para esse protótipo, os proxies interceptam a requisição obrigando o OF realizar o mapeamento para o proxy, duplicando o mapeamento para a origem, evidenciando o repasse de requisição para a origem por não ter armazenado o dado fisicamente em cache. Nos demais, o mapeamento interno, e o mapeamento para a origem ficou evidenciado. O código fonte completo do sistema operacional encontra-se no Anexo A.1.

4.5 Proxiamento Reverso com Balanceamento de Carga

De acordo com a descrição apresentada na Seção 3.2.5.3 da Página 80, a implementação do repasse de requisições e cacheamento físico são apresentados nesta seção. Por conta da alta compatibilidade entre o proxy e o load balance, foi escolhido uma implementação de *software* de terceiros, onde seus respectivos *daemons* são inicializados junto com a criação da rede.

4.5.1 Implementação do Proxy

A utilização do Proxy NGINX foi motivado pela sua simplicidade de implementação e por sua interoperabilidade com qualquer serviço web. Além disso, em *webservers* com arquitetura descentralizada, não teríamos a certeza sobre a caracterização do trabalho de pesquisa, uma vez que estaríamos colocando uma aplicação completa, ou seja, com vários arquivos de configuração espalhados pelo diretórios do S.O hospedeiro. Outra característica importante é a possibilidade de ser instanciado como processo em *background* e com várias instâncias diferentes no mesmo sistema, o que facilitou a configurações do load balance de backup como outro processo no S.O do Mininet.

NGINX foi desenvolvido por Igor Sysoev em 2002, tendo sua primeira versão pública disponível em 2004. Suas funcionalidades estendem-se a: Servidor Web, Proxy Reverso e Proxy Balanceador de Carga, com uma performance bem próxima ao do mais popular, o Apache, segundo Builtwith (2018). Em aplicações de alta performance, esse server pode alcançar milhares de conexões simultâneas através de combinações de módulo em tempo real. O autor Sato (2016) descreve alguns benefícios que o tornaram popular:

- Alto poder de respostas: Possibilidade de até 50.000 conexões simultâneas;
- Como proxy e balanceamento de carga: Front–end web via FastCGI ou HTTP consumindo 75% de CPU a menos por ser escrito em C;
- Servidor de e-mail POP3 e IMAP;
- Processo simples de instalação e configuração sem necessidade de *restart* após realização de alguma configuração.

A configuração implementada para atender os requisitos da pesquisa utilizou as configurações básicas da instalação, realizando poucas alterações no arquivo de configuração global, localizado em: ”***$ /etc/nginx/nginx.conf***“, e no arquivo de configuração do proxy localizado em: ”***$ /etc/nginx/sites-available/Default***“. Os dados contidos na configuração para o proxiamento das conexões, tiveram seu caminho incluído no arquivo global na diretiva ”http“ por meio de duas clausulas *include*, sendo:

- include /etc/nginx/conf.d/*.conf;
- include /etc/nginx/sites-enabled/*;

Assim, o *daemon* executou o mapeamento para o *webserver*, que no caso, é o próprio proxy reverso, criando automaticamente *links* simbólicos no diretório ***sites-enabled***. Seguindo a ordem de configurações utilizadas na pesquisa, a configuração global é apresentada por meio da Tabela 12. O arquivo de configuração do proxy encontra-se no B.2, ressaltando que ambos

os proxies, local ou em nuvem, utilizam da mesma grade configurações globais e virtual *hosts* alterando apenas o encaminhamento para os destinos.

Tabela 12 – Configuração Global para *Daemon* do Proxy Local ou na Nuvem

Diretiva	Parâmetro	Descrição
Root	worker_process 4	Informa que o servidor trabalhará com no mínimo 02 CPUs
Root	PID /run/nginx.pid	arquivo que armazena o pid do processo principal
Events	worker_connections	número máximo de conexões simultâneas por processo (worker_process * worker_connections)
html	proxy_cache_path	Define o caminho que o proxy fará o cache fisico no disco
	send_file on	não bloqueia a entrada e saída de dados do disco informando que cache não será em memória
	tcp_push on	usado em conjunto com send_file para enviar o cabeçalho de resposta de pacotes ao S.O.
	tcp_nodelay on	usado para transferência para o estado keepalive
	keepalive_timeout 65	tempo para manter a conexão com o cliente
	types_hash_max_size 2048	tamanho máximo da tabela de hash
	include /etc/nginx/mime.types	define do tipo MIME padrão para resposta. ativa uma extensa lista de formatos de dados permitidos
	default_type application/octet-stream	ativa a opção de stream de vídeo
	access log /var/log/nginx/access.log	realiza os logs de acesso ao proxy
	error_log /var/log/nginx/error.log;	realiza os logs de erros do proxy
	gzip on	ativa a compactção gzip mode
	gzip_disable "msie6"	bloqueia a navegação com o brownser IE6
	include /etc/nginx/conf.d/*.conf	inclui os caminhos para os arquivos de configuração
	include /etc/nginx/sites-enabled/*	inclui o caminho para os hosts virtuais

Por padrão, quando não se utilizam as configurações de servidores web Apache a configuração para os *hosts* virtuais pode ser feita diretamente no arquivo global "***nginx.conf***". Logo, preferimos utilizar a configuração de arquivos em seprado de modo a deixar o ambiente mais "elegante" para efetuar ajustes "*on the fly*" no NGINX. Essas configurações são obtidas por meio dos parâmetros "*include*" presente na diretiva "HTML" do arquivo global. Assim, as configurações para os *hosts* virtuais são apresentados na Tabela 13 conforme documentação oficial do sistema (NGINX.ORG, 2018).

Tabela 13 – Configuração do Virtual Host do Proxy Local e em Nuvem

Diretiva	Parâmetro	Descrição
upstream slicecloud	(least_conn, ip_hash, hash, least_time)	Algoritmos utilizados para funções de balanceamento de carga
	server(s) FQDN:porta	Define o grupo de servidores reais utilizados pelo proxy
server	listen 80;	Define a porta que será escutada pelo proxy
	access log off;	Desativa os logs de acesso
	expires max;	Define o tempo de expiração do cache
server location/*	proxy_cache (nome_da_zona_de_cache)	Aciona a Zona de Cache físico definida globalmente
	client_max_body_size,1m;	Tamanho Máximo do cabeçalho da requisição
	client_body_buffer_size 16k;	Define o tamanho do corpo da requisição em 16k (64 bits)
	proxy_send_timeout,90;	Define um tempo limite para transmitir uma resposta ao cliente
	proxy_read_timeout,90;	Define um tempo limite para ler uma resposta do proxy
	proxy_buffer_size,2k;	Define o tamanho do buffer usado para ler a primeira parte da resposta recebida do servidor com proxy (por padrão 4k ou 8k)
	proxy_buffers,16 4k;	Define o número e o tamanho dos buffers usados para ler uma resposta do servidor com proxy, para uma única conexão
	proxy_connect_timeout 30s;	Tempo limite para estabelecimento de uma conexão com o proxy
	proxy_pass http://slicecloud;	Define o protocolo e o endereço de um servidor com proxy e um URI opcional para o qual um local deve ser mapeado
	proxy_set_header,Host,$host;	Permite redefinir ou anexar campos ao cabeçalho da solicitação passado ao servidor com proxy. O valor pode conter texto, variáveis e suas combinações.
	proxy_set_header,X-Real-IP,$remote_addr;	É definido para o endereço IP do cliente para que o proxy possa tomar,decisões corretamente ou gerar log com base nesta informação.
	proxy_set_header,X-Forwarded-For $proxy_add_x_forwarded_for;	gera uma lista contendo os endereços IP de todos os servidores proxy. Por exemplo, a variável $proxy_add_x_forwarded assume o valor do cabeçalho X-Forwarded-For recuperando dados do requisitante

Em sua maioria os parâmetros adotaram o padrão para funcionamento da aplicação, mas de todos 03 (três) diretivas merecem atenção, sendo enumeradas logo abaixo:

1. ***Upstream SliceCloud/SliceLan***: O algoritmo de balanceamento utilizado será declarado acima do grupo de servidores que farão o *backend* para o proxy. Todos os algoritmos foram testados, sendo ***Least_conn*** o mais rápido.

2. ***Proxy_Cache_Zone***: Define a zona para de cache físico. Quando o proxy interceptar a requisição, ele fará a cópia do cache do *backend*, recuperando o arquivo caso não seja modificado por meio de um bit. A zona da nuvem fará *backup* do cache da Lan, caso assuma. E a zona da Lan, fará o *backup* da nuvem, enquanto estiver em produção.

3. ***Proxy_Pass http://SliceCloud ou SliceLan***: Repassa a requisição ao domínio assumida pelo proxy para o grupo de servidores finais ”*backend*“, disparando o algoritmo para executar o balanceamento.

4.5.2 Algoritmos de Balanceamento de Carga

Na Seção anterior, foi apresentada a grade de configuração utilizada para que o *daemon* do NGINX pudesse ser carregado pelo Host H6 no lado da Local e pelo Host H9 no lado da nuvem. A diretiva *upstream* funciona duplamente. Essa diretiva, por padrão ativa o algoritmo de função de balanceamento de carga Round Robin, e ainda traz a lista de servidores web que receberão o repasse de requisições pelo proxy reverso. Além disso, outros 03 (três) algoritmos de função de balanceamento foram testados. Sendo assim, essa Seção apresentará as funções e implementações de forma resumida dos algoritmos disponíveis.

A utilização de caches para requisições web traz um resultado esperado na maioria dos casos. Sendo que, essa pesquisa busca utilizá-los com cópias na nuvem e objetiva ter um bom desempenho mesmo em qualquer ambiente redes de conteúdo com SDN. As especificações de cada algoritmo são enumeradas logo abaixo:

1. **Roundrobin (RR)**: Esse é o método padrão e não precisa ser declarado explicitamente na diretiva ”upstream“. O método funciona através de atribuição de pesos ao grupo de servidores web por meio parâmetro ”weight“. Onde, o servidor que tiver o maior peso receberá o maior número de requisições;

2. **Least_conn**: Esse método na verdade é uma função para o algoritmo RR. A atribuição de peso para um determinado servidor é baseado na quantidade de requisições, ou seja, o server que no momento estiver com maior número de requisições por tempo médio, receberá um peso baixo, forçando o RR a mudar de servidor para atender as requisições, sucessivamente. Esta função também é conhecida como calculador *leastconn* para RR. O cálculo pode ser obtido através de uma simples ponderação de valor, sendo N.º de Conexões de um Servidor (N) divido pelo somatório do N.º de Conexões do grupo de servers, ou seja, $S(N) / \sum_{n=8}^{N}$, onde 8 é o peso padrão.

3. **IP-Hash**: Conhecido como método de mistura do IP de destino obtém os 03 (três) primeiros octetos do endereço IPv4 de clientes para calcular o valor do *hash*. Esse *hash* será dividido pelo peso total de servidores em funcionamento designando qual servidor irá receber o pedido. O valor do *hash* pode ser obtido pro meio do seguinte algoritmo:

```
ip_addr=(srcIP e Mask) hash_index=(ip_addr) + (ip_addr » 8) + ip_addr » 16) + (ip_addr » 24)
```

4. **Hash (*Generic Hash*)**: Uma chave é pré-definida combinando textos e variáveis para poder ser utilizado, ou, o que é mais genérico, a utilização de uma URI a partir de uma URL. Por exemplo, uma combinação de texto com IP e Porta.

5. **Least_Time**: O servidor receberá a conexão baseado em um cálculo médio de dois parâmetros: "*header*" ou "*last_byte*". A menor latência média e o menor número de conexões ativas farão com que o server receba a solicitação. Por exemplo: o SRVWEB01 tem 600 conexões ativas e peso=6; SRVWEB02 tem 400 conexões ativas e peso=4; SRVWEB03 tem 130 conexões ativas e peso=1. Logo, SRVWEB01 receberá a conexão porque mesmo com 600 conexões ativas, tem peso 6, então ocorrerá uma divisão (600 conexões / 6 pesos) = 100, ficando com menor latência.

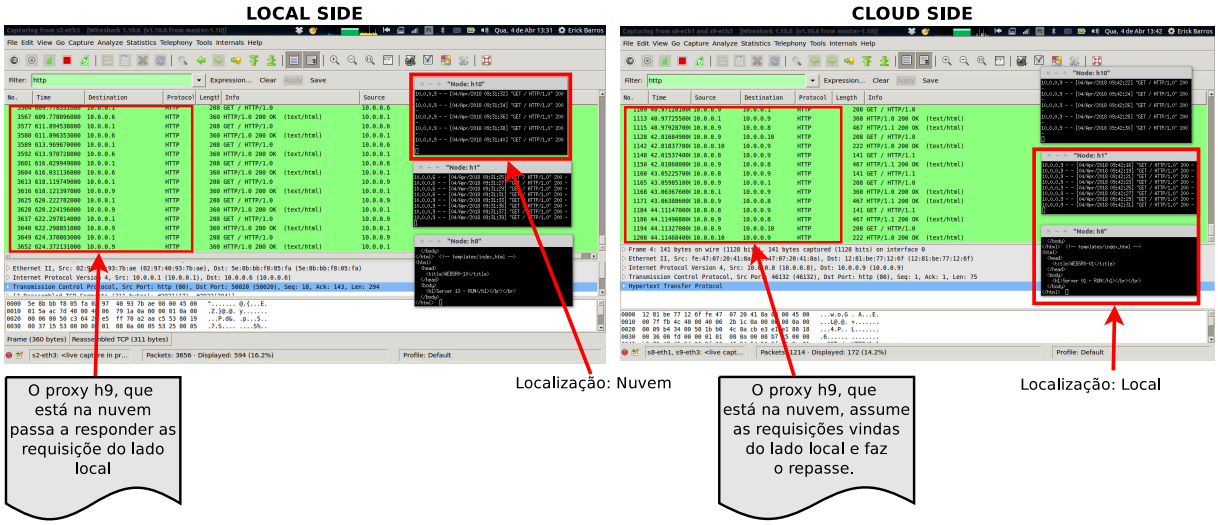


Figura 41 – Capturas de pacotes de tráfego HTTP entre proxies.

4.5.3 Alta Disponibilidade para o Load Balance

O sistema de Balanceamento de Carga foi implementado em duas camadas, para o caso de qualquer um dos Proxies Reversos fiquem indisponíveis, i.e., sem o *backend* Web, o Proxy Reverso Local ou da Nuvem possam assumir ambos os lados, mantendo o *backend* Web ativo a partir da nuvem ou a partir do lado local com todos os caches e replicações ativos.

Conforme apresentado na Seção 3.2.5.3 da Página 80, a redundância para o proxy reverso que administra as requisições no *backend* para os destinos finais (*WebServers*) se faz necessário. Isso porque caso o proxy fique off-line, todas as requisições ao domínio **http://example.com** ficarão comprometidas, e a nuvem não poderá assumir o lado local porque não receberá a informação que o proxy (MASTER) não estará mais disponível.

Para isso, a implementação da redundância para a o *backend* da nuvem foi implementada da seguinte forma: 1) Criando uma interface de túnel genérico virtual no *switch* S1 que está localizado no lado local; 2) Criando uma interface de túnel genérico virtual no *switch* S8 que

está localizado na nuvem. Como apresentado na Seção 2.3.3.2 da Página 48. A integração desses dois *switches* L(2) com funções de L(3) poderão ser obtidas através de um comando executado internamente durante a montagem da topologia. Os comandos para a junção desses *switches* são apresentados logo abaixo:

```
s1.cmd(''ovs-vsctl add-port s1 s1-gre1 - set interface s1-gre1 type=gre options:remote_ip=''+NODE2_IP)
```

O *switch* S1 através de seu *Command Line* invoca a ferramenta ovs-vsctl que cria uma porta virtual em S1, chamada de S1-GRE1, definindo seu tipo para GRE aceitando conexões remotas vindas do IP Público do Controlador da Nuvem. O mesmo ocorre para o *switch* S8 que está localizado na nuvem. O Encapsulamento de Roteamento Genérico (do inglês *Generic Routing Encapsulation (GRE)*) permite o tráfego do protocolo VRRP, ou seja, o mesmo protocolo utilizado pelo Keepalived para fazer redundância de proxies utilizando NGINX, o que permitiu a replicação das requisições para nuvem e consequentemente o cacheamento das mesmas, otimizando o tráfego de todo o sistema.

A configuração para o *daemon* do keepalived é relativamente simples e é feito no arquivo: ''*$ /etc/keeplived/keepalived.conf*''. A Seção 3.2.5.3 descreve o funcionamento do keepalived com o protocolo VRRP. A alta disponibilidade para o proxy reverso pode ser representado pela topologia mostrada na Figura 42.

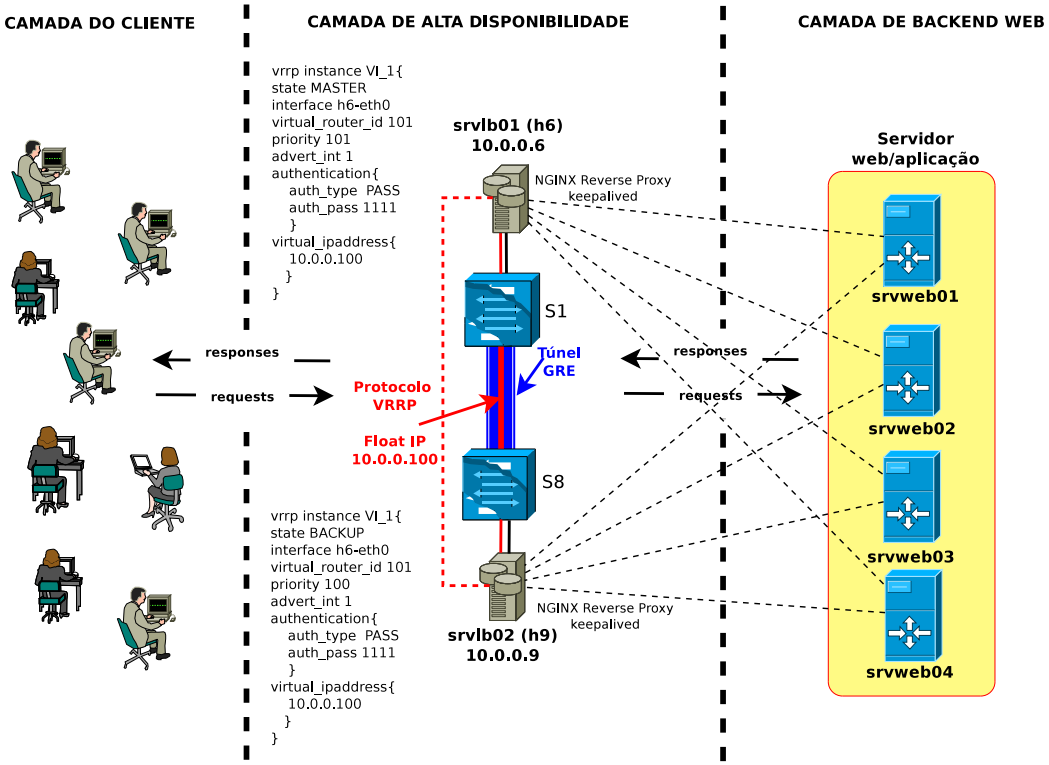


Figura 42 – Alta disponibilidade dos proxies reversos

Como pode ser visto na Figura 42, os *hosts* virtuais que funcionam como proxy na arquitetura são o **h6** e o **h9**, cada um tem seu próprio endereço IP, atribuído na inicialização da topologia. O *daemon* keepalived cria um adaptador de rede virtual como um extensão do adaptador real "eth0:vip". Enquanto o LB definido como MASTER estiver ativo, esse endereço de rede estará virtualizado em seu adaptador de rede, enquanto o LB de BACKUP fica se anunciando na rede para assumir caso o MASTER entre em falha, caso isso aconteça, esse mesmo IP "flutuará" para o BACKUP e ele assumirá as funções até que o MASTER fique novamente on-line. Quando o MASTER retornar, o IP flutuará novamente de volta para o MASTER que assumirá o controle. Isso somente é possível, por conta do túnel GRE estabelecido entre S1 e S8, onde o protocolo VRRP tráfega dentro do túnel através do IP público dos controladores. A Tabela 14 descreve os parâmetros de configuração.

Tabela 14 – Parâmetros de configuração do keepalived

Diretiva	Definição	Tipo
vrrp instance	identifica uma instância VRRP que definirá o bloco de configuração	
state	identifica o estado da instância em uso, se MASTER ou BACKUP	
interface	identifica a interface de rede que o *daemon* irá monitorar	string
virtual router ID	identifica qual ID do roteador VRRP a instância pertence	númerico
advert_int	específica o tempo em segundos para o anúncio na rede	númerico
authentication	identifica a autenticação VRRP definida para o bloco	númerico
auth_type	específica o tipo de autenticação se PASS ou AH	
auth_pass	específica o *password* da autenticação	string
virtual_ipaddress	específica o endereço IP virtual para o protocolo VRRP que será utilizado no bloco	string

Um detalhe que chama a atenção na grade de configuração da Tabela 14 é o tipo de autenticação. Isso demonstra que, além dos pacotes trafegarem pelo túnel GRE, ainda abre possibilidades de segurança para esse tráfego. O método *Authentication Header (AH)* pode ser utilizado quando há utilização de IPSec para o túnel estabelecido, enquanto o método PASS trafega a senha em texto simples pelo túnel. Para esta pesquisa o método PASS é utilizado por não objetivar mudanças no escopo por conta da autenticação do protocolo VRRP pelo túnel GRE. A Figura 43 mostra o VRRP em funcionamento na arquitetura.

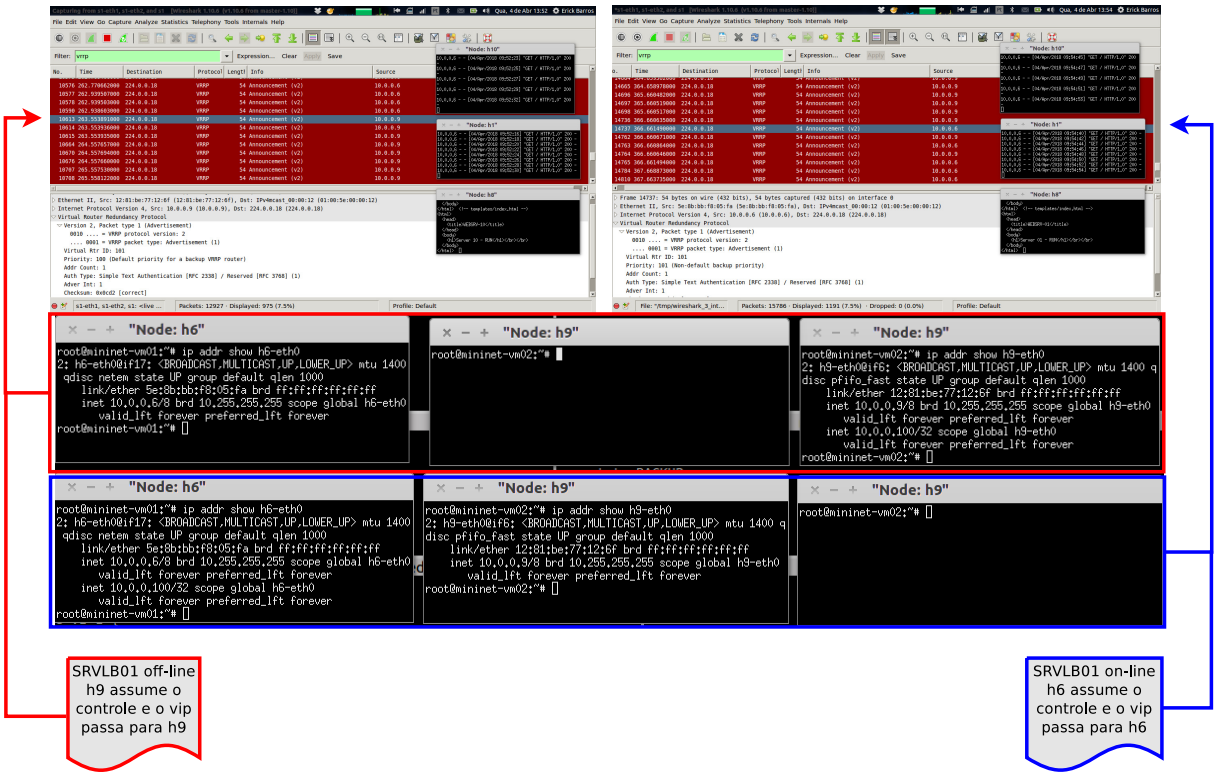


Figura 43 – Capturas de pacotes de tráfego VRRP entre proxies.

4.6 Implementação do Micro WebServer

Conforme descrito na Seção 3.2.6 da Página 82 a escolha pela utilização do Framework Flask Python, foi motivado por vários motivos. Além de seu recurso mínimo de execução, com pouco menos de 10 (dez) linhas de código fonte, ainda conta com um amplo conjunto de recursos que podem ser orquestrados para o desenvolvimento de grandes aplicações para a *World Wide Web (WWW)*.

O servidor web implementado, é uma instância de uma aplicação Flask, logo, todas as requisições originadas dos clientes que instanciam a aplicação para serem manipulados são passados do servidor web para processamento pelo protocolo Web Server Gateway Interface (WSGI). A inicialização da uma aplicação em Flask, necessita apenas de duas declarações, a primeira declaração, é a importação do objeto da classe Flask, por meio do "**from flask import Flask**"; a segunda declaração, é o argumento passado para o Construtor da Classe Flask informando o nome do módulo principal da aplicação. Esse argumento é atribuído a variável "**app**" que recebe o construtor através do argumento "**Flask(__name__)**". Essas declarações podem ser vistas logo abaixo.

```
from flask import Flask
app = Flask(__name__)
#...
```

Código 4.3 – Inicialização de aplicações em Flask

4.6.1 Funções de Visualização

As requisições (*requests*) que são feitas ao domínio "**example.com**" são encaminhadas aos servidores web. Essas requisições são passadas para a aplicação que indica o código a ser executado com base na URL solicitada. A partir dessa ação, um mapeamento entre a URL solicitada e a função executada na aplicação é realizado, esse evento é chamado de "Rota". Um rota é um "**decorator**" (SMITH et al., 2003) que é implementado no código para adicionar alguma função específica a um método já criado, sem alterar o código do método que se deseja adicionar tal funcionalidade. O Código 4.4 mostra a utilização de decoradores para implementar funcionalidade a uma função existente em nosso microwebserver SRVWEB01 instanciado por "h1".

```
#...
@app.route("/")
def index():
        message = "Server 01 - RUN"
        return render_template("index.html", message=message)
#...
```

Código 4.4 – Função adicional de Roteamento para requisições

Como pode ser observado no Código 4.4 , tem-se uma função de primeira classe "*def index():*". Essa função sem parâmetro possui uma variável "message" que recebe uma *string* informando o servidor em funcionamento. Em seguida, um retorno (*return*) é invocado para executar uma função interna "render_template" para carregar a pagina "index" do sistema, sendo que essa função não está declara no código porque trata-se de uma função interna ao decorador implementando. Logo, essa funcionalidade que fora adicionada pertence ao decorador "**app.route**". Esse decorador foi passado para função de primeira classe contendo um parâmetro do tipo *string* com o caminho que será acessado pela requisição (*request*), sendo assim, requisições diretamente ao domínio retornaram pela aplicação com o "index" do sistema. Esse tipo de configuração, permitiu a implementação das funções de endereçamento para arquivos de tamanhos diferentes e o cacheamento em memória das requisições que alcançarem o *endpoint* antes de serem interceptadas pelo Servidor de Conteúdo, denominado de **CACHE_MISS**. Um visualização interativa é apresentado na Figura 44.

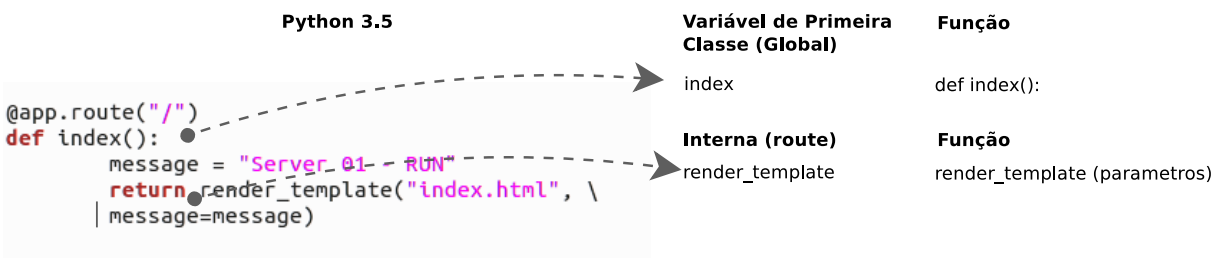


Figura 44 – Visualização de função de primeira classe

4.6.2 Funções de Obtenção de Conteúdo

As ideias que envolveram o desenvolvimento dessa pesquisa, inicialmente visavam a otimização de requisições apenas para endereços Web, vislumbrando a aceleração de respostas (*responses*) pelo cache dessas requisições. Mas como trata-se de redes de informação e conteúdo, ou seja ICNs, a implementação de acessos a conteúdos físicos encapsulando sua localização e distribuindo carga, foi realizada por meio da combinação de métodos de cache nos *endpoints*, utilizando os algoritmos presentes no proxy reverso, que geram *Hashes* para mapeamento em memória de arquivos físicos armazenados no disco.

Essas combinações permitiram uma configuração transparente, que foi chamado de microcaches. Esse cacheamento é a implementação de cache em memória (SIMPLE_CACHE), gerando um *hash* para cada tipo de conteúdo requisitado, de acordo com uma URI enviada, funcionando como um ponteiro para a memória. Esse *hash* assim que interpretado pelo proxy reverso, será copiado fisicamente para o disco, mas o mapeamento continuará em memória. Graças aos algoritmos de cache, a perda de performance pelo I/O de disco não onerou as requisições. Alguns ganhos podem ser listados abaixo:

- Cache de *Request/Reponse* espalhados geograficamente pela rede com omesmo conteúdo;
- Duplicação (*Backup*) desse Cache LOCAL e em NUVEM;
- Expiração de Cache forçando atualização do *hash*, CACHE_HIT;
- Otimização dos recursos de rede apenas pela replicação de servidores no domínio;
- Replicação do Conteúdo (Redundância);
- Alta disponibilidade pela replicação "**X4**" LOCAL e em NUVEM;
- Traffic Shapping Layer 7.

O Código 4.5 mostra a combinação de decoradores (*decorators*) com o método GET para cacheamento e obtenção de um arquivo estático encapsulando sua URI. Este microwebserver implementado para a pesquisa possui os principais métodos web, sendo: GET, POST, PUT e DELETE.

```
#...
@app.route("/static/images/125" ,methods=["GET"])
@cache.cached(timeout=30)
def staticImages125():
        with open("static/images/125.jpg", 'rb') as bites:
                return send_file(
                        io.BytesIO(bites.read()),
                        attachment_filename='125.jpg',
                        mimetype='image/jpg'
```

```
                    )
#...
```

Código 4.5 – Função para Obtenção de Conteúdo Binary I/O (*buffered stream*)

O trecho de Código 4.5 mostra uma função com decoradores atribuindo funções adicionais ao método por meio do "'**app.route**" "**cache.cached**". O primeiro decorador contém uma *string* com o caminho a ser executado pela aplicação, invocando a função "*def staticImage125()*" que invoca a função interna "send_file" da Classe Flask para transformar o conteúdo de imagem requisitado em *byte stream* e ser transportado no *response*; O segundo, realiza um SIMPLE_CACHE do conteúdo requisitado e estabelece um *timeout* de 30 segundos para o que mesmo expire e a função "*def staticImage125()*" tenha de ser novamente executar um novo cacheamento da requisição. A Figura 45 mostra de forma interativa a execução desse código. O Código fonte completo do sistema microwebserver está disponível no Apêndice A.3.

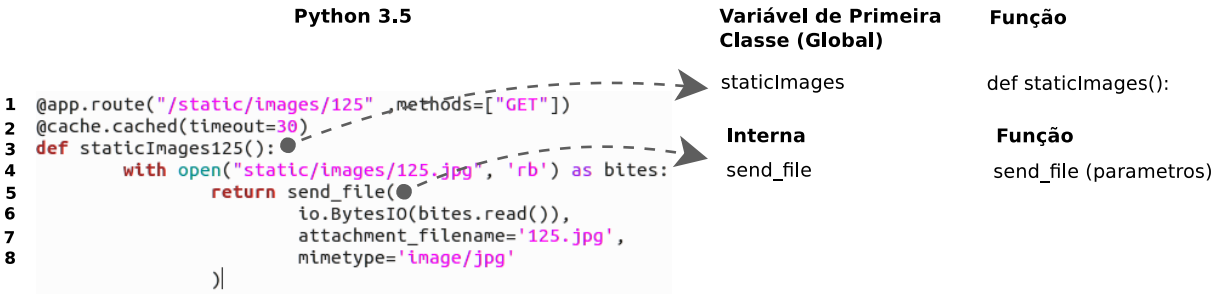


Figura 45 – Visualização interativa das funções adicionais de rota

A Figura 45 traz a implementação do método GET de uma requisição a uma conteúdo do sistema. Podemos ver na linha 2 (dois) o decorador para implementar a funcionalidade de cache para a função logo abaixo. Essa implementação é declarada da seguinte forma: "cache = Cache(config='CACHE_TYPE':'simple')". Um variável global cache Cache(config='CACHE_TYPE':'simple') recebe uma instância da classe Cache com o parâmetro CACHE_TYPE':'simple'. Esse parâmetro é o que ativa o cacheamento em memória para requisições para a URI definida em 1 (um).

Na linha uma 4 (quatro), uma operação de I/O Binário é iniciando para retornar a requisição para a chamada do método através de função "send_file". Essa função cria um fluxo de I/O Binário (*bufferd I/O*) para produzir objetos do tipo bytes. Esta categoria de *stream* pode ser utilizado para todos os tipos de dados não textuais, e quando o tratamento de dados de texto é necessário (PYTHON.ORG, 2018), ou seja, modificar a URI para que a aplicação possa carregar o arquivo físico em fluxo de bytes, e devolver a requisição (*response*) encapsulando o conteúdo solicitado na URL. Esse tipo de implementação permite que também seja gerado o *Hash* para o cache com base na *string* da variável bites que foi utilizada para ler os bytes do arquivo físico.

4.6.3 Cacheamento em Memória

Um decorador ”cache.cached(timeout=30)“ possui a funcionalidade que permite que uma requisição a uma determinada URL a qual ele pertença, ao retorno dos dados o mesmo será cacheado pela aplicação. A execução do cache pelo webserver foi descrito na Seção 3.2.6 da Página 82 e na Seção 3.3 da Paginá 83. O gerenciamento das funções de Cache pela aplicação pode ser controlado declarando a instância do Cache globalmente na inicialização da aplicação, logo, essa configuração pode ser alcançado de 2 (dois) modos:

1. **Durante a instanciação da Classe Cache**: cache = Cache(config=’CACHE_TYPE’: ’simple’).
2. **Pela chamada do método init_app**: cache.init_app(app, config=’CACHE_TYPE’: ’simple’)

Um terceiro modo pode ser utilizado, usando o padrão *Factory*, o que foi utilizado nesta pesquisa. Essa escolha se deu pela clareza de sua declaração.

```
cache = Cache(config={'CACHE_TYPE': 'simple'})
app = Flask(__name__)
cache.init_app(app)
```

4.7 Considerações Finais do Capítulo

Neste capítulo foram detalhadas as etapas para implementação do ambiente experimental. Foram descritos desde a contextualização das requisições executadas no domínio pertencente ao ambiente, implementação da topologia, inicialização de sistemas, configurações de repasse, cacheamento, entre outros.

A rede SDN foi instalada em duas máquinas virtuais ambos com o emulador Mininet Emulator 2.2.2. Além disso, o Framework Ryu SDN foi instalado no ambiente de modo a permitir que redes SDN fossem criadas pelo mininet e serem orquestrados por esse controlador, montando e desmontando fluxos automaticamente nos *switches* OF LOCAL e em NUVEM.

As topologias de rede foram implementadas por meio de *Scripts* Python para topologias customizadas, sendo a topologia em árvore para o lado LOCAL, e uma topologia linear para o lado da NUVEM. A topologia em árvore foi utilizada para que pudéssemos aproximar ainda mais, a arquitetura projetada para um cenário real.

O domínio constituído para o projeto, declarado como ”**example.com**“ e a resolução de nomes de todos os elementos da rede foram associados no arquivo de configuração do Bind, utilizando seu *daemon* em *background* para compor a resolução de nomes e endereços da rede.

O proxy reverso foi implementado com duas funções. A primeira, em suas funções

comuns de proxy transparente para repasse de conexões para servidores web no *endpoint*, ou seja, no destino final, acessando a aplicação web. A segunda, funções de *Load Balance* para o próprio servidor proxy, implementando a alta disponibilidade com suporte da nuvem.

Dois conjuntos de servidores web foram implementados com a linguagem Python, utilizando vários recursos de implementação, tais como: métodos globais para variáveis de primeira instância, ou seja, declaradas no ”main“ da aplicação. as funções desses métodos podem chamar métodos de outras classes de bibliotecas do Python, bem como, uso de decoradores para adicionar funcionalidades a estes métodos.

Por fim, a arquitetura foi montada plenamente funcional. Seus resultados serão apresentados no próximo capítulo.

5 Resultados Experimentais

Neste capítulo serão apresentados os resultados experimentais do desenvolvimento da pesquisa. Os dados foram obtidos a partir de requisições com processamento paralelo e saída de dados em arquivos físicos. Esses arquivos, posteriormente à finalização de cada experimento, foram tratados utilizando ferramentas específicas, tabulações, cálculos e geração de gráficos, expondo a análise dos resultados.

5.1 Cenário do Experimento

Este cenário utiliza uma adaptação idealizada através do autor Gondim (2016), que desenvolveu um protótipo de monitoramento de desempenho com uso de *middleboxes*, capturando dados da rede após execução de *scripts* pré-programados para obtenção dos resultados.

Assim, foram avaliados 3 (três) cenários distintos dos quais foram obtidos os resultados com um considerado grau de confiabilidade. Primeiramente, foram avaliados comportamentos referentes ao tráfego origem/destino de uma rede de computadores, tempo de resposta, perda, vazão, etc.

Depois foram realizados testes para validação do tunelamento. Apenas dados de perda foram avaliados neste momento, visto que vazões mínima e máxima, foram apresentados no primeiro cenário. Logo, latência, perda de pacotes, tempo médio entre falhas e tempo médio entre reparo, foram avaliados neste cenário, para validar a eficiência da integração com a nuvem para replicação de conteúdo e performance.

Por fim, o cenário que corresponde aos objetivos específicos desta pesquisa são apresentados. Aplicação de carga de tráfego HTTP por meio de URIs diretas e aleatórias, buscaram a otimização dos recursos da rede pelo de uso de caches de conteúdo, foram avaliados as possibilidades de controle de uma infraestrutura Local através da Nuvem, sem causar impactos significativos sendo apresentados e validados. Tempo de Requisição e Resposta, Taxa de Transferência, Tempo de Acesso com Cache e Sem Cache, Requisições Paralelas; todos esses Localmente e em Nuvem foram validados encerrando a parte experimental.

Ressalta-se neste que, todos os *scripts* de geração de tráfego são executados de forma sequencial, um por vez, pelo motivo de analisarmos os dados com maior segurança, a execução dos teste foi automatizado parcialmente para que não houvesse problemas que não foram percebidos imediatamente.

5.2 Ambiente Experimental

O ambiente para execução dos experimentos foi configurado em duas máquinas virtuais com Sistema Operacional Ubuntu 14.04.5 LTS, com 4 CPUs virtuais e 8 GB de memória RAM cada. Nestas máquinas foram instalados o emulador Mininet 2.2.2, emulando duas topologias distintas, sendo uma para o lado LOCAL e outra para a NUVEM, utilizando o SON da rede com Framework Ryu SDN v4.16 e Open vSwitch 1.3. Ainda nessas VMs, foram instalados *softwares* adicionais para execução dos experimentos, sendo:

- HTTPing 1.5.8 para realizar requisições HTTP e medir seus tempos de resposta, latência e taxa de transferência;
- IPERF 3.0.7 para obtenção da largura de banda alcançada e medição da perda de pacotes;
- ApacheBench V2.3 Rev.1528965 (RAHMEL, 2013) para aplicar a carga de requisições HTTP randômicas na rede;
- Python 3.5 and Flask Web Framework;
- Flask-Cache 1.3.2 uma biblioteca para suportar o cache da aplicação Python Flask.

5.3 Parametrização para Execução

5.3.1 Parâmetros para Validação da Rede e do Túnel GRE

Conforme descrito nas Seções anteriores, três cenários foram avaliados para compor os resultados da pesquisa. Neste ponto, os parâmetros para obtenção dos dados para validar o *dataplane* estabelecido é apresentado. Numa rede de computadores, deve-se pelo menos garantir que Round Trip Time (RTT) esteja dentro de níveis aceitáveis para que seja possível a avaliação de qualquer sistema que funcione sob ele. Sendo assim, a validação do plano de dados é obtida da seguinte forma:

1. Execução do comando Ping durante 600 segundos com a rede em repouso, ou seja, sem requisições HTTP trafegando no enlace. Esse teste nos dará um RTT médio embasando o teste com carga na rede.
2. Execução do HTTPing durante 600 segundos com parâmetros para obtenção do atraso e da largura de banda máxima alcançada.

Por conveniência e baseando-se no cenário do autor Xiulei et al. (2016), foi definido o tempo de 600 segundos por cada rodada de testes. Portanto, mensagens de Eco e Requisições Web utilizam esse tempo. Os dados do comando foram obtidos da documentação oficial

(DIE.NET, 2018b).

Nome: ping, ping6 – envia ICMP ECHO_REQUEST para um *host* na rede.

Descrição: utiliza um datagrama ICMP obrigatório tipo 8 por meio de um ECHO_REQUEST para extrair um ECHO_REPLY tipo 0 de um *host* ou *gateway* de rede.

Parâmetro utilizado: ping [-c count]. As mensagens ECHO_REQUEST encerram após a contagem terminar com a opção *deadline* embutida para esperar até o último ECHO_REPLY para terminar para entregar o relatório.

Tanto o Ping como o HTTPing possuem métricas estatísticas para apresentação dos resultados de sua execução, sendo: MIN, MÉDIA, MAX e MDEV (Latência), Quantidade de Conexões, Percentual de Perda e Tempo (ms) absoluto.

As características do HTTPing são as seguintes (DIE.NET, 2018a):

Nome: Mede a latência e a vazão máxima (*throughput*) de um servidor web.

Parâmetro utilizado: [-g url]: acessa a URL/I passada; [-c count]: contagem de requisições; [-G]: Realiza uma transferência de conteúdo completa; [-b]: Combinado com [-G] retorna a taxa de transferência; [-S]: Retorna a latência para cada requisição.

5.3.2 Parâmetros de execução para Validação do Protótipo

A parametrização da ferramenta AB Benchmarking da Apache é apresentada nesta seção. Por conveniência, foram definidas as sequências de execuções com a seguinte ordem: 1.000 *Requests*, 2.000 *Requests*, 4.000 *Requests*, 8.000 *Requests*, 16.000 *Requests* e 32.000 *Requests* para abranger no contexto geral, todos os tipos de requisições que ocorrem na Internet. Sendo assim, todas as sequências de testes do sistema executaram as 32.000 requisições na ordem de 600 segundos, embasando o conjunto de dados para a validação da rede. A Tabela 15 apresenta os parâmetros utilizados na execução.

Tabela 15 – Parêmtros para Requisições HTTP

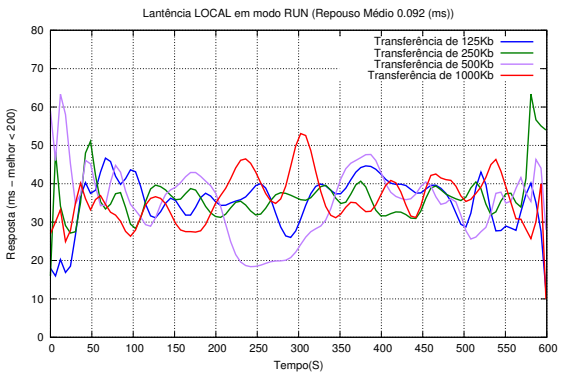
Parâmetros	Descrição
Path do documento URL/URI	example.com/images/125–1000
Tamanho (KB)	125,250,500,1000
Nível de Concorrência	100 Conexões Paralelas
Tempo de execução do Teste (s)	Média de 600 seconds
Protocolo de Transmissão	HTTP
Falha de Requisições	Número de Falhas
Total Tranferido	Max. 32 GB
Topologia de Rede	Árvore e Linear
Método de Execução	Sequencial
Tipo de Cache	LeastConn em Zona de Disco Cache (CACHE_DIR)

Dois exemplos do comando para geração de carga na rede pode ser visto em seguida:

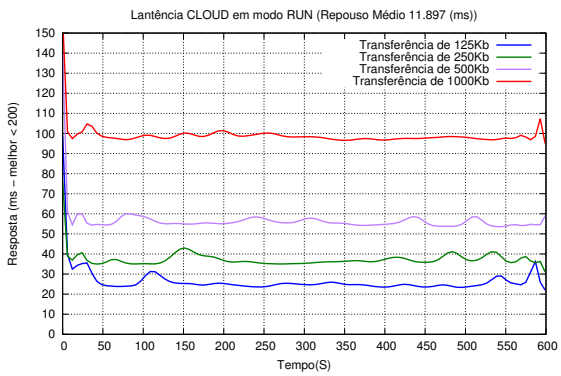
1. `$ ab -c 100 -n 1000 a 32000 http://example.com/static/images/125-1000 - requisições diretas ao domínio.`
2. `$ cat myurls.txt | ’’ab -c 100 -n 32000 {}‘‘ - Exec. Randômico.`

5.4 Validação da Rede

Para esta validação foram executadas duas sequências de testes para cada ambiente (*slice*) da arquitetura. As execuções ocorreram dos *hosts* **h8** e **h4** que estão localizados na parte mais inferior da rede, de acordo com a topologia geral da Figura 40 da Página 97. A Figura 46 (a) e (b) apresentam seus resultados.



(a) Echo_Requests Local



(b) Echo_Requests para Nuvem

Figura 46 – Echo Requests e Reply Ping e Httping

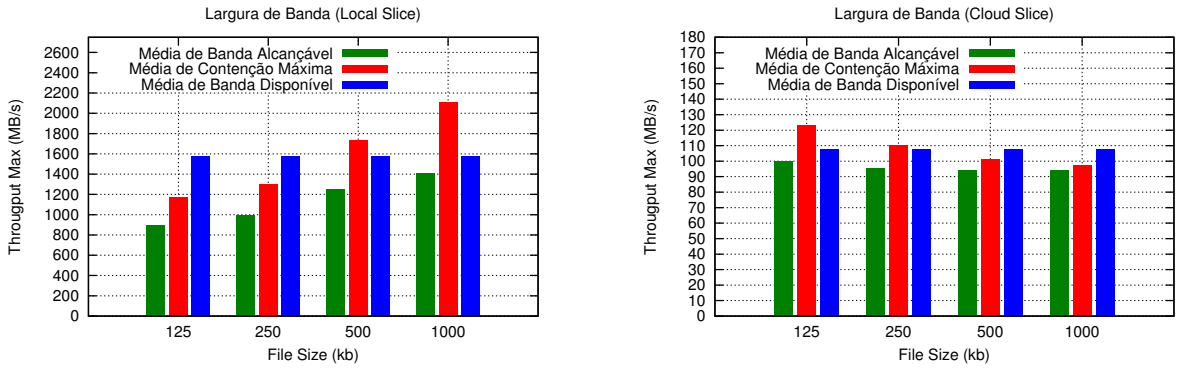
Se observarmos no lado A(Local), temos uma variação entre 20(ms) e 60(ms), onde na sequência da contagem, os números ficaram variando em torno de 30(ms) e 50(ms) durante toda a execução. O tempo médio de atraso obtido com a rede em repouso foi de 0.098(ms), o que nos leva a entender que a oscilação apresentada se dá por conta da topologia em árvore utilizada no lado Local. Um vez que, os *hosts* **h8** e **h4** estão na parte mais inferior da rede. Mas, ainda assim, com a rede em execução, foram obtidos resultados aceitáveis segundo Brito (2013). O sistema foi adequado para que as requisições atinjam diretamente os servidores finais, com o proxy reverso fazendo apenas o repasse de requisições HTTPing.

Observando o lavo B(Nuvem), temos uma variação na ordem de 20(ms) para arquivos de 125 Kb e de 100(ms) para arquivos de 1000 Kb, o que aponta para a perda de desempenho por conta da transferência pelo túnel. O tempo médio de atraso obtidos com a rede em repouso para requisições na nuvem foi de 11.897(ms), aumentando a latência para requisições em direção a nuvem consideravelmente. Ainda assim, foram obtidos valores aceitáveis para uso de aplicações em rede nessa arquitetura. A Tabela 16 mostra os dados estatísticos.

Tabela 16 – Resultados Estatísticos de Execução e RTT em modo RUN

— example.domain.local ping statistics —				
Performed	**Transmitted**	**Received**	**Loss (%)**	**Time (ms)**
Local	650	650	0	649070
Cloud	650	650	0	650072
RTT	**Min**	**Max**	**Avg**	**Mdev**
Local	0.094	0.128	4.458	0.179
Cloud	0.100	11.897	20.217	6.180

Continuando com a validação da rede, para garantir que haverá largura de banda suficiente para utilização de uma ICN, duas sequências de testes randômicos foram executados. Seguido o padrão distribuído mostrado na Seção 3.4.1.2 da Página 86. Foram executados requisições aleatórias no domínio, requisitando conteúdos de tamanho variado de 125 kb a 1000 kb, obtendo o relatório do HTTPing em seguida. As requisições ocorreram dos *hosts* **h8** e **h4** que localizados na parte mais inferior da rede, de acordo com a topologia geral da Figura 40 da Página 97. A Figura 47 apresenta os resultados dessa medição.



(a) Echo_Requests Local

(b) Echo_Requests para Nuvem

Figura 47 – Echo Requests e Reply Ping e Httping

Observando os gráficos da Figura 47 temos 3 (três) indicadores que são descritos em seguida:

1. **Banda Alcançável**: refere-se a valores que são alcançados dado uma transferência completa entre origem e destino;
2. **Contenção Máxima**: refere-se a valores que são alcançados sem a presença de tráfego, ou seja, somente uma requisição sem transferência completa ao destino;
3. **Banda Disponível**: refere-se a valores que são alcançados e garantidos pelo sistema, ou seja, independente de *overhead* causados por processamento, memória, *hardware*, essa banda está garantida.

Assim, temos que na Figura 46(a), como não há qualquer controle de banda, para permitir que o sistema atinja seus valores máximos. Adotemos a contenção máxima para efeitos de largura do enlace. Sendo assim, temos velocidades entre 900 Mbps para arquivos de 125kb e 1400 Mbps para arquivos de 1000kb, analisando a partir do último, temos uma disponibilidade de enlace de 67% para transferência de conteúdo.

Observando na Figura 46(b), sabendo que o enlace do Túnel não ultrapassa 1Gbps e adotando a contenção máxima de 120 Mbps para arquivos de 125kb e 90 Mbps de banda alcançável para arquivos de 1000kb, podemos projetar a disponibilidade do enlace para a nuvem na ordem de 75% para transferência de conteúdo. A Tabela 17 mostra os dados estatísticos.

Tabela 17 – Resultados Estatísticos de Execução RTT e HTTP

— example.com/static/images HTTPing statistics —								
Data	Local Slice				Cloud Slice			
RTT	**Min**	**Max**	**Avg**	**SD**	**Min**	**Max**	**Avg**	**SD**
125	11.8	252.5	41.1	42.18	20.2	99.5	26.5	9.12
250	13.0	507.6	48.3	45.64	31.4	97.5	37.7	8.58
500	13.8	445.0	46.8	53.41	49.8	140.6	56.5	8.60
1000	16.1	453.4	52.6	55.09	92.7	157.6	99.0	7.21
Speed (Kbps)	**Min**	**Max**	**Avg**	**SD**	**Min**	**Max**	**Avg**	**SD**
125	506	145781	36320	35267	8283	15352	12470	966.35
250	503	102736	47415	43734	6937	13798	11872	574,69
500	1092	216728	79939	62005	10050	12594	11728	328,93
1000	2233	263072	110624	79255	10962	12158	11728	174,29

5.5 Validação do Tunelamento Genérico (GRE)

Neste teste, foram executados dois experimentos para validarmos a aplicabilidade do tunelamento de Open vSwitches na transferência de conteúdo pela Internet, de acordo com a topologia geral apresentada na Figura 40 da Página 97. O primeiro teste, uma carga na rede foi realizada do proxy h6 para o proxy h9 objetivando uma execução de tráfego de rede de ponta a ponta pelos proxies, neste teste, apenas o cliente envia e o servidor responde criando um *stdout report* para um arquivo de texto. No segundo teste, o parâmetro ”-d“ é adicionado e o cliente h6 envia um carga na rede e o servidor também aplica uma carga executando um teste Bidirecional de ponta a ponta, verificando envio e recebimento de pacotes em modo *fullduplex*. O servidor IPERF3 iniciado no lado da nuvem, e o cliente IPERF3 iniciado pelo proxy no lado local geram tráfego para dentro do túnel durante 600 segundos.

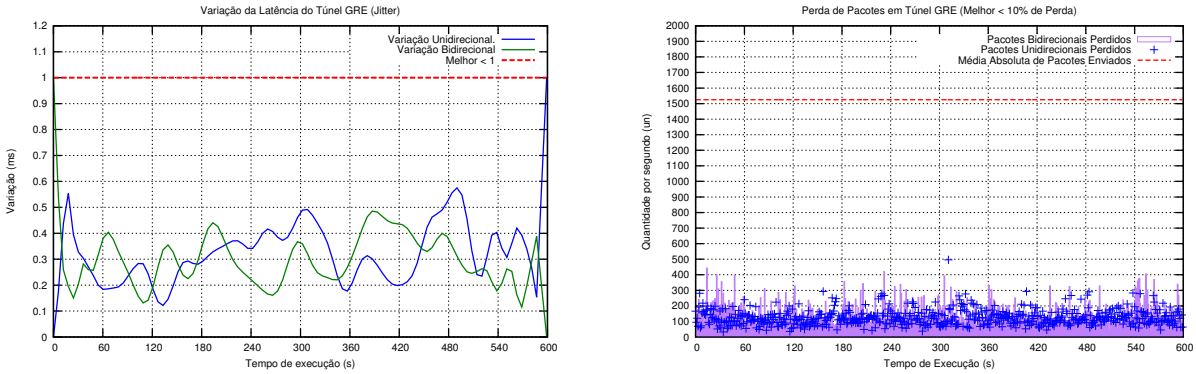
Os parâmetros utilizados pelo cliente e servidor IPERF3 são descritos: **Servidor**: -s [server]: define a inicialização como server; -p [port]: define a porta que irá escutar/acessar;

Cliente: -c [client] + [IP destino]: define a inicialização como cliente e IP que irá requisitar; -u [udp]: utiliza o protocolo UDP para medir o jitter; -d [dualtest]: executa com teste bidirecional; -b [band]: define uma banda para execução em Kbytes, Megabytes e Gigabytes.

As configurações foram as seguintes:

1. Servidor iniciado em h9 com o comando: ”iperf3 -s -p 12000“;
2. Cliente iniciado em h6 aplica a carga na rede com o comando: ”iperf3 -c 10.0.0.9 -u -p 12000 -b 100m“;
3. Cliente iniciado em h6 aplica a carga na rede com o comando: ”iperf3 -c 10.0.0.9 -u -p 12000 -d -b 100m“ para testes bidirecionais.

A Figura 48, apresenta o resultado dos experimentos para análise da variação da latência, Jitter. Essa medição de variação nos garantirá se as conexões ao *backend* terão um mínimo de estabilidade e o sistema não ficará emitindo retransmissões.



(a) Variação do Atraso (b) Perda de Pacotes

Figura 48 – Variação do Atraso e Perda de Pacotes em Túnel GRE

Observando a Figura 48a, temos uma variação ocorrendo na ordem de 0.150 ms e 0.580, ocorrendo numa máxima aos 480 segundos de experimento. O gráfico apresenta a oscilação por conta da latência do próprio controlador, mas o gráfico ainda apontou para os padrões permitidos, ou seja, *jitter* abaixo de 1, mostrando que o estabelecimento de conexões TCP foram estáveis.

Observando a Figura 48b, temos que perda de pacotes existente se mantém na ordem máxima aproximadamente de 200 pacotes perdidos para cada 1500 pacotes enviados. A configuração ideal de acordo com a variação da latência, seria uma perda inferior a 1%. Esse resultado poderá talvez evidenciar o próprio *overhead* produzido pela Internet, porque os conteúdos distribuídos poderiam estar armazenados em diferentes arquiteturas e enlaces, mas para as redes programáveis podem evidenciar um aumento de latência do controlador.

Além disso, como o túnel GRE encapsula o pacote em um cabeçalho novo, acaba que a

transferência máxima de segmento pelo túnel não funcionará corretamente com MTUs de 1500, logo, a MTU das interfaces de rede dos proxies, foram ajustadas para 1476 bytes, para que os pacotes possam passar pelo túnel. Como o tráfego não aponta para a FLAG (*Don't Fragment*–DF) ativa, entendemos que essa alta perda de pacotes ocorre pelo descarte dos mesmos pelos *switches*. Mas não houve retransmissões nem erros de transmissão. A Tabela 18 mostra os resultados dos experimentos. Em termos percentuais a perda de pacotes se manteve próximo aos 10%. A aplicação de um QoS no túnel possa melhorar essa perda, mais isso não foi testado.

Tabela 18 – Resultados Estatísticos das operações de carga com IPERF3

– example.com IPERF3 statistics –					
Mode	**Range (s)**	**Transf. (GB)**	**Bandwith (Mbps)**	**Jitter (ms)**	**Loss/Total/Loss %**
Unidirecional	0.00-600.08	6.98	100	0.750	75237 / 915366 / 8.2%
Bidirecional	0.00-600.06	6.86	100	0.970	85973 / 915275 / 9.3%

Por fim, conclui-se nesta seção que o tunelamento GRE é funcional para os objetivos da pesquisa, uma vez que, a transferência pelo túnel não apresenta erros significativos que venham a prejudicar o andamento da pesquisa. A arquitetura aponta para testes de latência no controlador, o que pode ser feitos em trabalhos futuros.

5.6 Validação das Requisições HTTP - Otimização do Sistema

Por fim, nesta seção o experimento alto da pesquisa é realizado. O gerenciamento é feito de acordo com os algoritmos de *load balance* utilizados no proxy reverso e os microcaches por meio de decoradores da aplicação. Apesar dos autores Badea et al. (2014) discordarem do uso dessa abordagem em ICNs baseadas em NDO, Round_Robin (RR), Least_Conn e IP_Hash foram utilizados para desenvolver a estratégia proposta e validar as operações de cache.

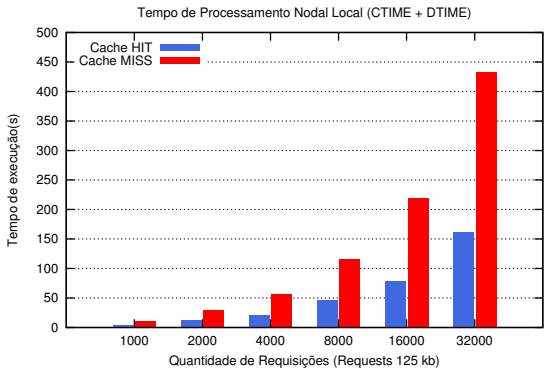
Um experimento de proporções mínimas foi utilizado para definir o algoritmo de controle das requisições. A Tabela 19 mostra os resultados obtidos balizando o uso do Least_Conn para os experimentos HTTP e os parâmetros descritos na Tabela 15 da Seção 5.3.2.

Tabela 19 – Comparação dos Algoritmos de Balanceamento de Carga

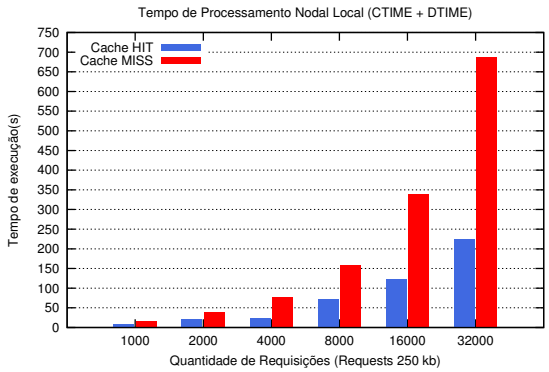
Algoritmos de Balanceamento de Carga			
	Round_Robin	Least_Conn	IP_Hash
Time 1000 Request(s)	61.216	59.711	66.299
Requests per sec(s)	18.56	17.66	15.77
Time per Request(ms)	61.21	59.71	66.29
Transfer Rate (Gbps)	6.72	6.86	7.54

5.6.1 Requisições HTTP Cacheadas e Não Cacheadas - *Local Slice*

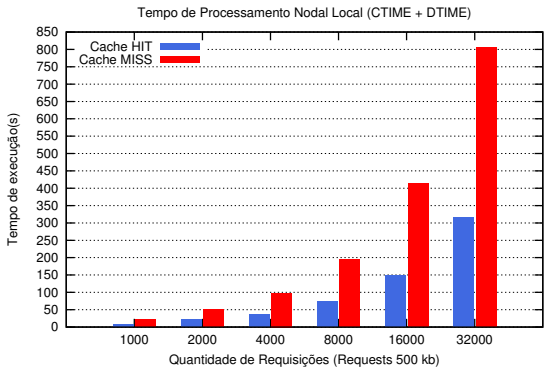
A Figura 49 apresenta os resultados de 32.000 requisições aos conteúdos disponíveis no sistema. Essas requisições significam o tempo total de requisições (*Total Requests*) em função do número total de requisições paralelas (*Total Response Time*), ambos roteados e não roteados para o cache. Em outras palavras, o CTIME é igual ao tempo para estabelecer a conexão (Latência), mais o DTIME, tempo de processamento da requisição. Os resultados mostraram que as respostas de solicitação respondidas pelo Cache (CACHE_HIT) otimizaram os acessos ao *endpoint* (CACHE_MISS). Este comportamento acontece devido ao cacheamento em disco realizado pelo proxy reverso e mapeamento em um bit do cache em memória, encurtando o caminho entre origem/destino.



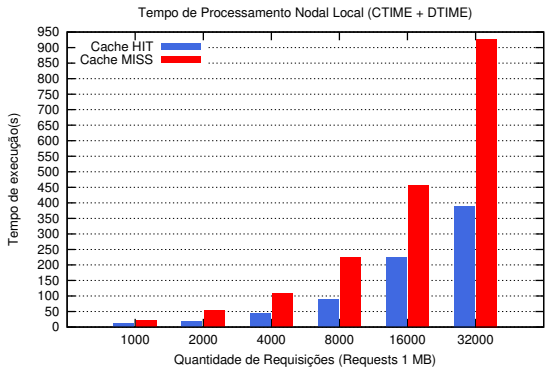
(a) Requisições a conteúdos de 125kb



(b) Requisições a conteúdos de 250kb



(c) Requisições a conteúdos de 500kb



(d) Requisições a conteúdos de 1000kb

Figura 49 – Nível de Concorrência de 100 Conexões Paralelas entre h4 e h8

Para o lado local, observemos os resultados contidos nos gráficos da Figura 49(d) para as 32.000 requisições para arquivos de 1000kb. Dois indicadores contém informações importantes acerca da otimização de requisições. O tempo gasto para execução dos testes (*Time Taken for Tests*) realizados para 32.000 requisições com o cache ativo (CACHE_HITS) foi de 389.474 segundos, enquanto requisições respondidas diretamente pelo *backend* foram executas em 927.204 segundos. Uma redução de 42% com transferência completa de conteúdo, seja do cache ou do *endpoint*.

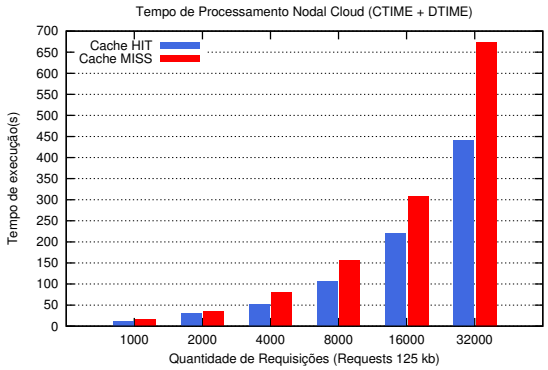
A largura de banda alcançável, medido pelo *Transfer Rate* foi de 79.54 Mbps para requisições com o cache (CACHE_HITS) ,enquanto 33.25 Mbps de largura de banda alcançável para requisições que não atingiram o cache (CACHE_MISS), apontando uma liberação na largura da banda de 41%, saltando de 33,25 Mbps para 79,54 Mbps de banda alcançável. A Tabela 20 mostra o resultado para conteúdos de 1000kb.

Tabela 20 – Resultados Estatísticos de Requisições de Conteúdo

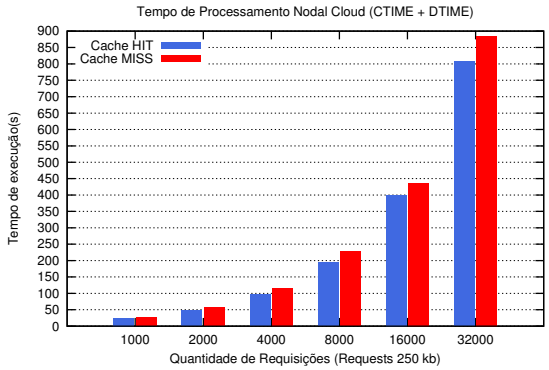
–AB Benckmarking statistics report - 1000kb – 32.000 requests/responses–		
Reports	**CACHE HITS**	**CACHE MISS**
Time Taken Requests (s)	389.474	927.204
Complete Request	31.998	32.000
Failed Requests	2	0
Total Transferred (GB)	31.56	31.57
Time per Requests (ms), mean all	12.171	28.97
Transfer Rate (Mbps)	79.54	32.25

5.6.2 Requisições HTTP Cacheadas e Não Cacheadas - *Cloud Slice*

A Figura 50 apresenta os resultados de 32.000 requisições com o controle da rede passado para a nuvem. No qual o total de requisições e o tempo de processamento CTIME + DTIME são avaliados, constituindo o tempo nodal do processamento de requisições. O tempo gasto para realização de 32.000 requisições com o cache cache (CACHE_HITS) ativo foram de 2746.872 segundos, enquanto requisições respondidas diretamente pelo *backend* da nuvem levaram 2959.749 segundos para completar os testes. Uma redução de 7,19% de requisições HTTP com transferência completa de conteúdo, seja do cache ou do *endpoint*.

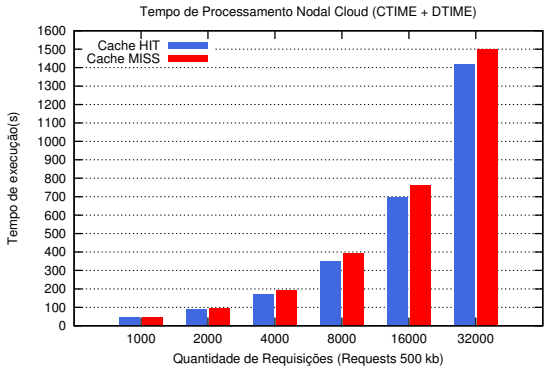


(a) Requisições a conteúdos de 125kb

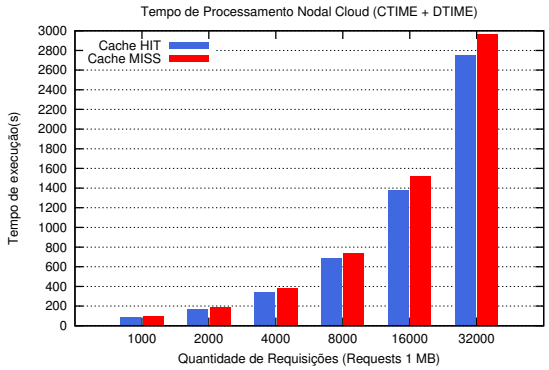


(b) Requisições a conteúdos de 250kb

Figura 50 – Nível de Concorrência de 100 Conexões Paralelas de h4 e h8



(a) Requisições a conteúdos de 500kb



(b) Requisições a conteúdos de 1000kb

Figura 51 – Nível de Concorrência de 100 Conexões Paralelas de h4 e h8

A largura de banda alcançável, medido pelo *Transfer Rate* foi de 11.22 Mbps com requisições com o cache (CACHE_HITS) ativo, enquanto 10.41 Mbps de largura de banda alcançável para requisições diretas no *endpoint* (CACHE_MISS), apontando uma liberação na largura da banda de 7,21%, saltando de 10,41 Mbps para 11,22 Mbps de banda alcançável. A Tabela 21 mostra o resultado para conteúdos de 1000kb.

Tabela 21 – Resultados Estatísticos de Requisições de Conteúdo

–AB Benckmarking statistics report - 1000kb – 32.000 requests/responses–		
Reports	**CACHE HITS**	**CACHE MISS**
Time Taken Requests (s)	2746.872	2959.749
Complete Request	32.000	32.000
Failed Requests	0	0
Total Transferred (GB)	31.57	31.57
Time per Requests (ms), mean all	85.84	92.49
Transfer Rate (Mbps)	11.22	10.41

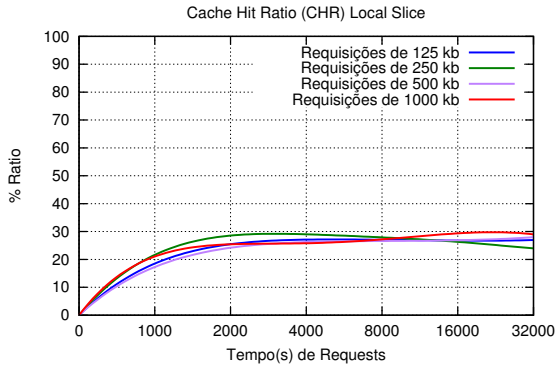
5.6.3 Avaliação do Ganho de Performance

Nesta seção, serão mostrados os ganhos percentuais da otimização de requisições HTTP pelo uso de caches. É fato que o uso de caches já é comumente utilizado, mas esta pesquisa utilizou-se desse recurso para redes de conteúdo, com respostas (*responses*) aleatórias, randômicos, replicados localmente, replicados em nuvem e sobre uma rede SDN. A Figura 52 apresenta os ganhos com o uso de caches para duas variáveis muito importantes, Cache Hit Ratio (CHR) e o Byte Hit Ratio (BHR).

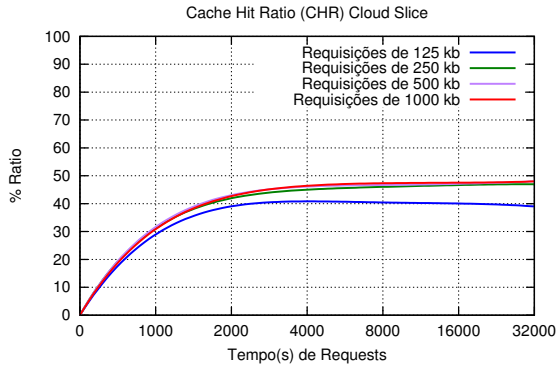
O CHR é obtido para avaliar a otimização do tempo de resposta de uma requisição, ou seja, avalia o ganho do sistema com as requisições sendo distribuídas entre o cache (proxy reverso) e o endpoint (*backend*). Para obter esses valores usamos a seguinte fórmula:

CACHE_HITS/(CACHE_HITS + CACHES_MISS) * 100

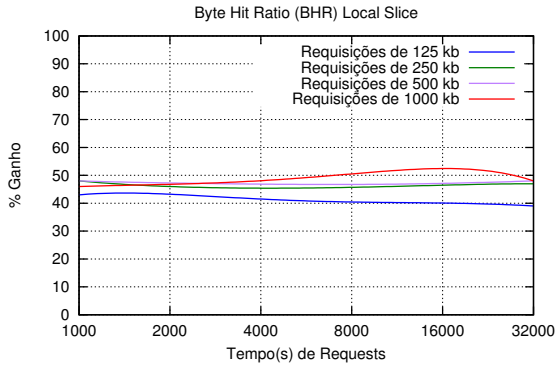
Mesmo com um status de "Cold Cache", caches frios, a otimização do sistema foi efetiva no lado local ficando entre 20% e 30%, ou seja, muitas conexões ainda passaram para os servidores web. Ajustar o tempo de expiração para menor poderá satisfazer este cenário, mas a otimização foi garantida. Já no lado da nuvem, um status de "Warm Cache"pode ser observado, já que a otimização do tempo ficou próximo aos 50% (OSSOWSKI, 2015). Mesmo passando pelo túnel na saída do lado local e na entrada da nuvem, se o conteúdo não tivesse expirado a resposta já retornaria do proxy da nuvem. A otimização para a taxa de transferência foi praticamente a mesma. A Figura 52 mostra os resultados.



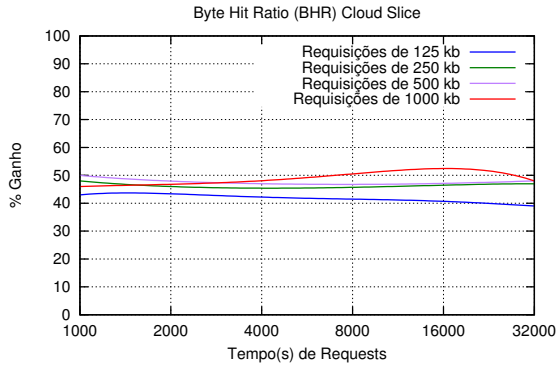
(a) Otimização da taxa de tempo local



(b) Otimização da taxa de tempo da nuvem



(c) Otimização da taxa de transferência local



(d) Otimização da taxa de transferência da nuvem

Figura 52 – Proporção de Otimização do Tempo de Resposta e Transferência

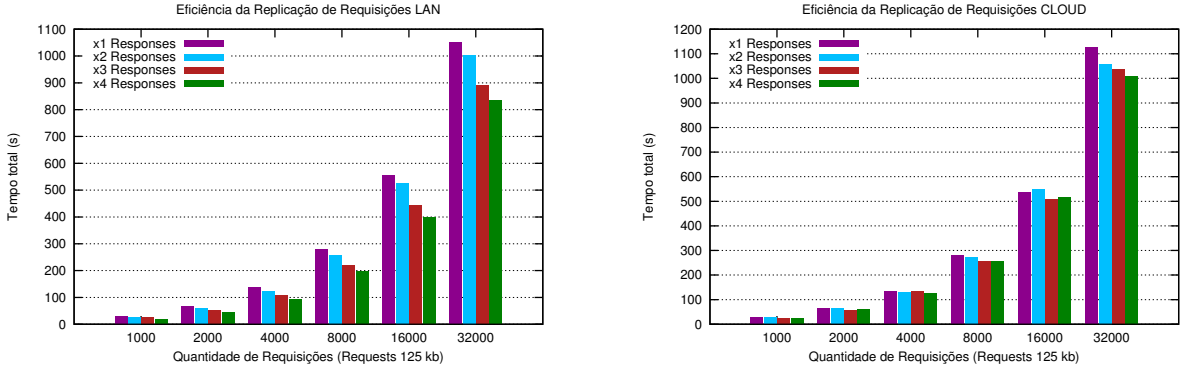
5.6.4 Eficiência da Replicação

Nesta seção, foram avaliados os aspectos acerca do cache da aplicação dos *endpoints* no *backend*, ou seja, o cache de aplicação nos *webservers*. Para que fosse possível tal avaliação, o cacheamento do proxy reverso reverso foi desativado, transformando em proxy transparente. Não há necessidade de avaliar todo o conteúdo do domínio example.com, uma vez que a regra que se aplica para um, vale para todos. Por isso o experimento foi executado somente em conteúdos de

125kb para ter retorno mais rápido dos dados.

Observando a Figura 53, nas colunas de 32.000 requisições, tomando para avaliação as requisições respondidas com um servidor respondendo apenas, ”X1 Responses“, e a coluna com 4 (quatro servidores) respondendo, ”X4 Responses“ apresentamos os resultados.

No lado local, as 32.000 requisições ”X1“ levaram 1050,501 segundos, que equivalem a 17 min e 50 seg. Replicando essa resposta ”X4“, temos 834,074 segundos, que equivalem a 14 min e 30 seg. Portanto só com a replicação o sistema obteve um ganho de 20,60%



(a) Otimização da taxa de tempo local

(b) Otimização da taxa de tempo da nuvem

Figura 53 – Proporção de Otimização do Tempo de Resposta

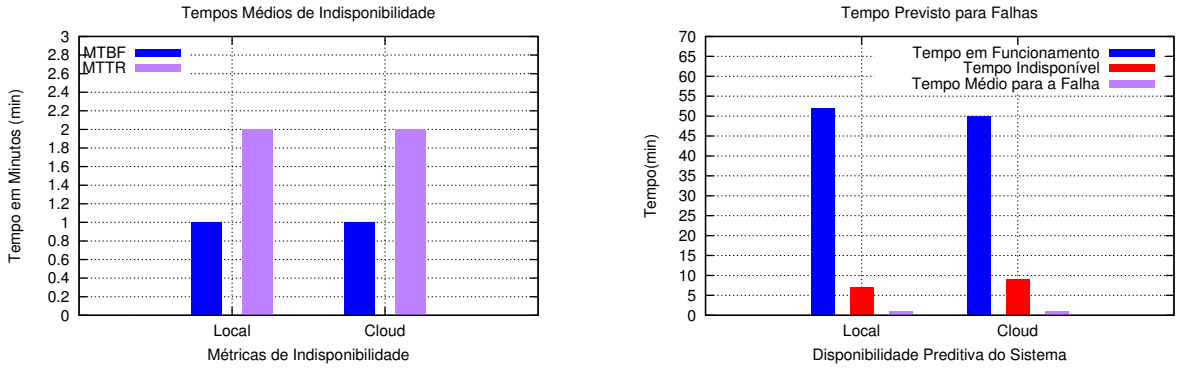
Já no lado da nuvem, 32.000 requisições ”X1“ *responses* foram obtidas e 1125,030 segundos, que equivalem a 19 min e 15 seg. Replicando essa resposta ”X4“, temos 1009,560 segundos, que equivalem a 17 min e 22 seg. Portanto, a replicação obteve um ganho de 10,26%

5.6.5 Disponibilidade do Sistema

Para finalizar a validação da rede, apresentamos a disponibilidade da arquitetura, por meio de simulação controlada. Esse experimento foi programado para 60 minutos, onde a cada 60 segundos, uma falha é aplicada ao sistema forçando-o requisitar a nuvem para assumir o controle, não permitindo que o *backend* local e em nuvem fiquem indisponíveis. Esse experimento resultou em 27 falhas no lado local e 26 falhas na nuvem. Destaca-se que todas as requisições continuaram a ser respondidas pelo cache, e ainda, transparentes para os requisitantes h4 e h8. Três métricas de disponibilidade foram observadas, sendo:

- ***Mean Time Between Failures (MTBF)***: verifica o tempo médio de uma falha para outra. Obtém-se por meio da fórmula: (TEMPO DE PREVISTO - TEMPO PARADO)/QTD. FALHA;
- ***Mean Time To Repair (MTTR)***: verifica o tempo médio para o sistema se recupera (o lado local assumir novamente) Obtém-se por meio da fórmula: TEMPO PREVISTO/QTD. FALHA

- ***Mean Time To Failed (MTTF)***: verifica o tempo médio para que a falha aconteça (*availability*). Obtém-se por meio da fórmula: MTTF = MTBF/(MTBF+MTTR)



(a) Tempo de Indisponibilidade do Sistema

(b) Tempo de Funcionamento do Sistema

Figura 54 – Variação do Atraso e Perda de Pacotes em Túnel GRE

Observando a Figura 54a, temos os tempos de MTBF e MTTR respectivamente. No lado ”Local“, O MTBF apesenta o tempo de 1,92. Isso significa que a cada 2 minutos e 32 segundos um falha ocorrerá no local e a nuvem assumirá. O MTTR apresentou um resultado de 2 minutos e 22 segundos, logo, o sistema local se recuperará após esse tempo e o lado local voltará a assumir o controle.

Observando a Figura 54b, o MTTF é apresentado. Para este, foi necessário colocarmos a visualização do funcionamento, onde no lado local, o sistema esteve 52:28 minutos funcionando e 07:32 minutos parado, obtendo 0,46 de disponibilidade. Enquanto, em nuvem ficou 50:27 minutos funcionando e 09:33 minutos parado, obtendo 0,45 de disponibilidade.

Diante destes, entende-se que o sistema apresenta uma disponibilidade regular, uma vez que passa dos 91,86% se levarmos em conta a alta disponibilidade da nuvem. A Tabela 22, apresenta o resultado completo .

Tabela 22 – Avaliação Preditiva da Arquitetura (60 Min Exec.)

Indicadores	**Local**	**Nuvem**
Tempo Previsto de Operação (min)	01:00:00	01:00:00
Tempo Indisponível	00:07:32	00:09:33
Tempo de Sistema em Produção	00:52:28	00:50:27
MTBF	1,92	1,92
MTTR	2,22	2,30
MTTF (Disponibilidade)	0,4637	0,4549
% de disponibilidade	46,37 %	45,49 %
% de disponibilidade total da arquitetura	91,86%	
% de indisponibilidade da arquitetura	8,14%	

Com todas as análises observadas nas Seções anteriores, conclui-se a avaliação da arquitetura de rede programável SDN para ICN. Os dados obtidos foram consistentes nos três cenários utilizados, com informações importantes acerca de modelos de arquitetura flexíveis e programáveis bem como, mecanismos de otimização de infraestruturas de redes de computadores e sistemas distribuídos. A próxima seção apresenta a conclusão do trabalho, seus desafios e perspectivas para novos desafios.

6 Conclusão e Trabalhos Futuros

Esta pesquisa realizou o desenvolvimento de uma arquitetura de rede definida por *software* (SDN) para suportar o uso de sistemas de conteúdo tipo ICN. O trabalho busca a otimização de requisições e respostas a conteúdos disponíveis na internet, minimizando a degradação dos enlaces pela duplicação da requisição ao mesmo conteúdo pelo mesmo enlace. A arquitetura também prevê, a replicação do conteúdo entre publicadores (*publishers*) e ainda ter uma cópia em nuvem para dar suporte em caso de alto consumo de recursos ou indisponibilidade de informação, garantindo que uma rede remota seja capaz de assumir o controle, sem perda significativa para os clientes que acessam os conteúdos disponíveis no plano de dados. Como os controladores SDN tem total controle sob à rede, podendo ainda, acessar interfaces externas (públicas), tudo isso por meio da programabilidade de dispositivos SDN, sugere-se uma análise sobre a capacidade de suporte remoto a partir da integração de dois lados de uma plano de dados único.

A partir do estudo e modelagem da arquitetura, foi desenvolvido um protótipo ICN com replicação em nuvens privadas. Este protótipo é capaz de disponibilizar conteúdo de forma descentralizada, replicando suas informações através de outros publicadores programados previamente e distribuídos de forma geográfica no ambiente. O protótipo, é capaz de realizar o cacheamento em memória de uma requisição direto na origem transformando a URI em um *hash* relacionado com um endereço de memória. Caso essas requisições persistam, com o proxy reverso interceptando cada requisição, um gravação de cache em disco será executada, e passará a responder pelas requisições dos clientes. O lado remoto (em nuvem) assumirá o controle em caso de indisponibilidade de qualquer publicador do lado local, ou seja, no protótipo temos quatro *webservers* em cada lado, assim, se o primeiro do lado local falhar, o primeiro do lado da nuvem passará a responder as requisições, e assim sucessivamente, do mesmo modo de cacheamento.

Na implementação do protótipo, foram desenvolvidas APIs para as seguintes partes: uma para a topologia de rede local, uma para topologia de rede da nuvem, quatro publicadores para o lado local, quatro publicadores para o lado da nuvem; Um sistema operacional do tipo *learning switch* foi ajustado para ser usado pelo controlador SDN; Como os controladores possuem acesso direto ao *user space*, três *daemons* de terceiros foram executados no ambiente, sendo eles: *daemon* para resolução de nomes e endereços IP, *daemon* para balanceamento de carga, *daemon* para alta disponibilidade; Por fim, um tunelamento através de OpenVSwitch foi estabelecido conectando o plano de dados local com o plano de dados da nuvem.

6.1 Contribuições Científicas

O protótipo foi avaliado sob três cenários. O primeiro para validação dos enlaces. O segundo para validação da interligação local/remoto. O terceiro, para validação das requisições e do cache. Produzindo um total de oito avaliações. Em todos os experimentos, o protótipo obteve resultados que possibilitam o seu uso para otimização de redes e utilização para ICNs. Esses experimentos demonstraram que esse modelo pode ser usado para outras implementações.

Diante dos aspectos descritos na Seção 1.2, da Página 23, as contribuições da pesquisa são enumeradas logo abaixo:

1. Descentralização do controle da rede, habilitando e desabilitando recursos de forma automática e centralizada pelo *Controller*, seja local ou remoto;
2. Mapeamento automático entre origem/destino criando diferentes enlaces para cada requisição desfazendo-os quando encerrado a conexão (*Packet-in handler*);
3. Capacidade de operar sob uma latência variando de 0.098 ms a 11.897 ms. Esse resultado é consistente;
4. Largura de banda garantida para o lado local entre 900 Mbps e 1400 Mbps, isto é, da máxima, 67% funcionou efetivamente;
5. Largura de banda garantida para o lado da nuvem entre 90 Mbps e 120 Mbps, isto é, da máxima, 75% funcionou efetivamente;
6. Perda de pacotes no túnel abaixo de 10%, impactando minimamente no contexto geral, visto que de 7.0 GB de conteúdo randômico transferido, apenas 8.2% (local) e 9.3% (nuvem assumindo) foram realmente necessários serem retransmitidos, ainda assim, imperceptível para os clientes;
7. Distribuição de requisições com o algoritmo Least_Conn permitiu a redução nos acessos finais no lado local na ordem de 42%, elevando a banda para quase 80 Mbps de *download*;
8. Distribuição de requisições com o algoritmo Least_Conn permitiu a redução nos acessos finais no lado da nuvem na ordem de 7.19%, elevando a banda para 11.22 Mbps de *download*. O que torna o resultado consistente;
9. Ganho de 30% no Tempo Total da Requisição (Atraso + Processamento) no lado local (CHR);
10. Ganho de 30 a 40% no Tempo Total da Requisição (Atraso + Processamento) no lado da nuvem (CHR);
11. Disponibilidade de sistema para o caso de falhas na ordem de 91.86%.

6.2 Artigos Publicados e Submissões Futuras

Nesta seção, a Tabela 23 apresenta os artigos publicados durante o andamento da pesquisa bem como, um planejamento em potencial mostrado na Tabela 24 de submissões futuras que encontram-se em andamento. Essas contribuições foram alcançadas na medida que o projeto de pesquisa foi se consolidando.

Tabela 23 – Artigos Publicados em Anais de Eventos e *Journals*

Nº	Evento	Artigo	Autores	Qualis CC/Int
01	WETICE'2017 21-23 June Pozna, Poland	A Programmable Network Architecture for Information Centric Network using Data Replication in Private Clouds	Erick B. Nascimento Edward David Moreno Douglas D. J. de Macedo	CC B1
02	CONINFA'2017 23-27 Outubro Paulo Afonso-BA	Avaliação de Redes SDN utilizando Metodologia Experimental	Erick B. Nascimento Methanias C. R. Júnior Aricio M. da Silva Douglas D. J. de Macedo	INT B2
03	IJGUC'2018 March'18	Cache Replication for Information Centric Networks through Programmable Networks	Erick B. Nascimento Edward David Moreno Douglas D. J. de Macedo	CC B2
04	ISCC'2018 25-28 June Natal-RN, Brazil	Evaluation of Cache for Bandwith Optimization in ICN Through Software Define Networks	Erick B. Nascimento Douglas D. J. de Macedo Edward David Moreno Luis C. E. de Bona Miriam A. M. Capretz	CC A2

Tabela 24 – Planejamento para Publicações Futuras

Nº	Evento	Tema Provisório	Autores	Qualis CC
01	WSCAD'2018 01-03 Outubro São Paulo, SP, Brasil	Algoritmos de Cache para Otimização de ICNs usando Software Defined Network	Erick B. Nascimento Edward David Moreno Douglas D. J. de Macedo	CC B3
02	WSCAD-CTD'2018 01-03 Outubro São Paulo, SP, Brasil	Arquitetura de Rede Programável para Information Centric Network – ICN com Replicação de Conteúdo em Private Clouds	Erick B. Nascimento Douglas D. J. de Macedo Edward David Moreno	CC B3
03	UCC'2018 17-20 December Zurich, Suisse	ICNs Optimization through Memcached Replicated	Erick B. Nascimento Edward David Moreno Douglas D. J. de Macedo	CC A2
04	UCC'2018 17-20 December Zurich, Suisse	Scalability of SDN Memcached to High Operating of the Cloud and Fog Computing	Erick B. Nascimento Douglas D. J. de Macedo Edward David Moreno	CC A2
05	A definir	Implementação de Fila de Mensagens (*Message Queue*) para troca de informações em ICNs	Erick B. Nascimento Douglas D. J. de Macedo Edward David Moreno	CC
06	A definir	Implementação de *FileSharding* para ICNs com SDN	Erick B. Nascimento Douglas D. J. de Macedo Edward David Moreno	CC

6.3 Dificuldades Encontradas

Nesta seção, serão descritas algumas dificuldades encontradas que somente foram percebidas, durante o ponto alto dos experimentos, na hora de validar a arquitetura. Enumerou-se alguns deles, sendo:

1. As Figuras 46a e 48a, apontaram uma oscilação no envio de pacotes ICMP, oscilando bastante em ambos os lados. Essas mensagens foram enviadas em tempo de execução, com o ambiente com carga na rede, o que demonstra uma alta latência do *controller* para topologias em árvore.
2. Implementação de protocolos de vídeo RTP e RSTP, avaliando a distribuição e cacheamento. Não houve tempo para implementação para esses protocolos;
3. Comparação de túnel GRE com VxLAN para definição de qual seria utilizado;
4. Utilização de outras tecnologias de cache, uma vez que, alguns autores classificam Round Robin, LRU e Least_Conn, como métodos já testados.

6.4 Trabalhos Futuros

Por fim, algumas possibilidades serão apresentadas para evolução deste trabalho de pesquisa, buscando a continuação do trabalho de pesquisa e de seu aperfeiçoamento.

1. Estudo para possibilidade de escalonamento sob demanda, o que poderia otimizar ainda mais as requisições;
2. Estudo para possibilidade de utilização de uma rede *mesh*;
3. Aplicar o *benchmarking* do controlador local para tratamento da oscilação apresentada nas mensagens ICMP;
4. Verificar possibilidades de adaptação da arquitetura para uso de novas técnicas de cache, tais como: *MemCached*, Fila de Mensagens (*Message Queue*) e *Sharding*;
5. Realização dos mesmos experimentos com as técnicas mencionadas no item anterior, uma vez que, *Mamcached* pode ser usado em nosso proxy reverso;
6. Utilização de técnicas para melhoramento do CACHE_MISS, como alocação dinâmica, análise de pontos por função e Algoritmos adaptativos.

Referências

ABDULLAHI, I.; ARIF, S.; HASSAN, S. Survey on caching approaches in information centric networking. *Journal of Network and Computer Applications*, Elsevier, v. 56, p. 48–59, 2015. Citado na página 39.

AHLGREN, B. et al. A survey of information-centric networking. *IEEE Communications Magazine*, IEEE, v. 50, n. 7, p. 26–36, 2012. Citado 3 vezes nas páginas 9, 36 e 37.

AHLGREN, B. et al. Second netinf architecture description. *4WARD EU FP7 Project, Deliverable D-6.2 v2. 0*, 2010. Citado na página 37.

AIVALIOTIS, D. *Mastering Nginx*. [S.l.]: Packt Publishing Ltd, 2016. Citado na página 78.

AKBAR, M. S. et al. Information-centric networks: categorizations, challenges, and classifications. In: IEEE. *Wireless and Optical Communication Conference (WOCC), 2014 23rd*. [S.l.], 2014. p. 1–5. Citado 3 vezes nas páginas 9, 38 e 39.

AKONJANG, O. Sane: A protection architecture for enterprise networks. Citeseer, 2007. Citado na página 27.

ANAN, M. et al. Empowering networking research and experimentation through software-defined networking. *Journal of Network and Computer Applications*, Elsevier, v. 70, p. 140–155, 2016. Citado na página 41.

ARBETTU, R. K. et al. Security analysis of opendaylight, onos, rosemary and ryu sdn controllers. In: IEEE. *Telecommunications Network Strategy and Planning Symposium (Networks), 2016 17th International*. [S.l.], 2016. p. 37–44. Citado 2 vezes nas páginas 9 e 75.

AUBRY, E.; SILVERSTON, T.; CHRISMENT, I. Srsc: Sdn-based routing scheme for ccn. In: IEEE. *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. [S.l.], 2015. p. 1–5. Citado 4 vezes nas páginas 54, 57, 59 e 60.

AZODOLMOLKY, S. *Software Defined Networking with OpenFlow*. [S.l.]: Packt Publishing Ltd, 2013. Citado na página 30.

BADEA, R. A. et al. A critical perspective towards ccn. In: IEEE. *Communications (COMM), 2014 10th International Conference on*. [S.l.], 2014. p. 1–6. Citado na página 122.

BERDE, P. et al. Onos: towards an open, distributed sdn os. In: ACM. *Proceedings of the third workshop on Hot topics in software defined networking*. [S.l.], 2014. p. 1–6. Citado na página 75.

BRITO, S. H. B. *Interpretação dos Resultados do Ping*. 2013. <http://labcisco.blogspot.com.br/2013/05/interpretacao-dos-resultados-do-ping.html>. [Online; accessado em 12-Abril-2018]. Citado na página 118.

BUILTWITH. *Web Server Usage Statistics*. 2018. <https://trends.builtwith.com/web-server>. [Online; accessed 02-April-2018]. Citado na página 102.

CAESAR, M. et al. Design and implementation of a routing control platform. In: USENIX ASSOCIATION. *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. [S.l.], 2005. p. 15–28. Citado na página 27.

CARVALHO, M. S. R. M. de. *A trajetória da Internet no Brasil: do surgimento das redes de computadores à instituição dos mecanismos de governança*. Tese (Doutorado) — UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2006. Citado na página 21.

CASSEN, A. Keepalived: Health checking for lvs & high availability. *URL http://www. linuxvirtualserver. org*, 2002. Citado na página 80.

CENTRAL, S. *What are SDN Northbound APIs?* 2016. <https://www.sdxcentral.com/sdn/definitions/north-bound-interfaces-api/>. [Online; accessed 26-November-2016]. Citado na página 30.

CHI, X. et al. Web load balance and cache optimization design based nginx under high-concurrency environment. In: IEEE. *Digital Manufacturing and Automation (ICDMA), 2012 Third International Conference on*. [S.l.], 2012. p. 1029–1032. Citado na página 78.

CISCO, C. V. N. I. Global mobile data traffic forecast update, 2013–2018. *white paper*, 2014. Citado na página 20.

CISCOSYSTEMS. *DNS Best Practices*. 2018. <https://www.cisco.com/c/en/us/about/security-center/dns-best-practices.html>. [Online; accessed 28-January-2018]. Citado 2 vezes nas páginas 9 e 71.

DESAI, A. et al. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering*, Citeseer, v. 2, n. 3, p. 222–225, 2013. Citado 2 vezes nas páginas 9 e 66.

DIE.NET. *HTTPing - Linux man page*. 2018. <https://linux.die.net/man/1/httping>. [Online; accessado em 12-Abril-2018]. Citado na página 117.

DIE.NET. *Ping - Linux man page*. 2018. <https://linux.die.net/man/8/ping>. [Online; accessado em 12-Abril-2018]. Citado na página 117.

EDELMAN, J. *Are there alternatives to the OpenFlow protocol?* 2013. <http://searchsdn.techtarget.com/answer/Are-there-alternatives-to-the-OpenFlow-protocol>. [Online; accessed 26-November-2016]. Citado na página 33.

FALAGAS, M. E. et al. Comparison of pubmed, scopus, web of science, and google scholar: strengths and weaknesses. *The FASEB journal*, FASEB, v. 22, n. 2, p. 338–342, 2008. Citado na página 52.

FP7. *The PERSUIT Project*. 2017. <http://www.fp7-pursuit.eu/PursuitWeb/>. [Online; accessed 20-October-2017]. Citado na página 37.

GAO, S. et al. Scalable control plane for intra-domain communication in software defined information centric networking. *Future Generation Computer Systems*, Elsevier, v. 56, p. 110–120, 2016. Citado na página 23.

GHODSI, A. et al. Naming in content-oriented architectures. In: ACM. *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*. [S.l.], 2011. p. 1–6. Citado 3 vezes nas páginas 56, 59 e 60.

GIL, A. C. Como elaborar projetos de pesquisa. *São Paulo*, v. 5, n. 61, p. 16–17, 2002. Citado na página 61.

GONDIM, E. B. Monitoramento de desempenho com middleboxes em redes definidas por software. 2016. Citado na página 115.

GRINBERG, M. *Flask web development: developing web applications with python*. [S.l.]: "O'Reilly Media, Inc.", 2014. Citado na página 82.

GROUP, N. W. et al. Rfc 2616 hypertext transfer protocol–http/1.1. *R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee*, 1999. Citado na página 87.

GUDE, N. et al. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 3, p. 105–110, 2008. Citado na página 27.

HAKIRI, A.; GOKHALE, A. Data-centric publish/subscribe routing middleware for realizing proactive overlay software-defined networking. In: ACM. *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. [S.l.], 2016. p. 246–257. Citado 4 vezes nas páginas 54, 55, 59 e 60.

HALEPLIDIS, E. et al. *Software-defined networking (SDN): Layers and architecture terminology*. [S.l.], 2015. Citado na página 66.

HELLER, B. et al. Elastictree: Saving energy in data center networks. In: *Nsdi*. [S.l.: s.n.], 2010. v. 10, p. 249–264. Citado na página 41.

HINDEN, R. et al. Virtual router redundancy protocol (vrrp); rfc3768. txt. *IETF Standard, Internet Engineering Task Force, IETF, CH*, p. 0000–0003, 2004. Citado na página 81.

JACOBSON, V. et al. Networking named content. In: ACM. *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. [S.l.], 2009. p. 1–12. Citado 2 vezes nas páginas 36 e 39.

JAMJOOM, H.; WILLIAMS, D.; SHARMA, U. Don't call them middleboxes, call them middlepipes. In: ACM. *Proceedings of the third workshop on Hot topics in software defined networking*. [S.l.], 2014. p. 19–24. Citado na página 28.

JONES, T. G. *Reverse Http*. 2014. <http://reversehttp.net/>. [Online; accessed 02-March-2018]. Citado 2 vezes nas páginas 9 e 83.

KAUR, K.; SINGH, J.; GHUMMAN, N. S. Mininet as software defined networking testing platform. In: *International Conference on Communication, Computing & Systems (ICCCS)*. [S.l.: s.n.], 2014. p. 139–42. Citado 2 vezes nas páginas 67 e 69.

KHONDOKER, R. et al. Feature-based comparison and selection of software defined networking (sdn) controllers. In: IEEE. *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*. [S.l.], 2014. p. 1–7. Citado 2 vezes nas páginas 11 e 77.

KITCHENHAM, B. Procedures for performing systematic reviews. *Keele, UK, Keele University*, v. 33, n. 2004, p. 1–26, 2004. Citado na página 49.

KIVITY, A. et al. kvm: the linux virtual machine monitor. In: *Proceedings of the Linux symposium*. [S.l.: s.n.], 2007. v. 1, p. 225–230. Citado na página 80.

KONRAD, M. *Simple caching with Python pickle and decorators*. 2016. <https://datascience.blog.wzb.eu/2016/08/12/a-tip-for-the-impatient-simple-caching-with-python-pickle-and-decorators/>. [Online; accessado em 02-Abril-2018]. Citado na página 83.

KOPONEN, T. et al. A data-oriented (and beyond) network architecture. In: ACM. *ACM SIGCOMM Computer Communication Review*. [S.l.], 2007. v. 37, n. 4, p. 181–192. Citado 2 vezes nas páginas 37 e 54.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, IEEE, v. 103, n. 1, p. 14–76, 2015. Citado 4 vezes nas páginas 21, 22, 27 e 31.

KUMAR, R. et al. Apache cloudstack: Open source infrastructure as a service cloud computing platform. *Proceedings of the International Journal of advancement in Engineering technology, Management and Applied Science*, p. 111–116, 2014. Citado na página 42.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: ACM. *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. [S.l.], 2010. p. 19. Citado na página 66.

LE, V. D. et al. Microservice-based architecture for the nrdc. In: IEEE. *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*. [S.l.], 2015. p. 1659–1664. Citado na página 82.

LEINWAND, A.; CONROY, K. F. *Network Management (2Nd Ed.): A Practical Perspective*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996. ISBN 0-201-60999-1. Citado 4 vezes nas páginas 10, 84, 85 e 86.

LI, D. et al. A survey of network update in sdn. *Frontiers of Computer Science*, Springer, v. 11, n. 1, p. 4–12, 2017. Citado na página 20.

LI, L. et al. Design patterns and extensibility of rest api for networking applications. *IEEE Transactions on Network and Service Management*, IEEE, v. 13, n. 1, p. 154–167, 2016. Citado na página 30.

LI, W.; MENG, W.; KWOK, L. F. A survey on openflow-based software defined networks: Security challenges and countermeasures. *Journal of Network and Computer Applications*, Elsevier, v. 68, p. 126–139, 2016. Citado na página 28.

LUNDH, F. *Python standard library*. [S.l.]: "O'Reilly Media, Inc.", 2001. Citado na página 82.

LUO, H. et al. Efficient integration of software defined networking and information-centric networking with color. In: IEEE. *2014 IEEE Global Communications Conference*. [S.l.], 2014. p. 1962–1967. Citado 3 vezes nas páginas 57, 59 e 60.

MARCONI, M. d. A.; LAKATOS, E. M. *Fundamentos de metodologia científica*. [S.l.]: 5. ed.-São Paulo: Atlas, 2003. Citado 2 vezes nas páginas 61 e 62.

MASOUDI, R.; GHAFFARI, A. Software defined networks: A survey. *Journal of Network and Computer Applications*, Elsevier, v. 67, p. 1–25, 2016. Citado na página 28.

MEALLING, M.; DENENBERG, R. *Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations*. [S.l.], 2002. Citado na página 79.

MEDVED, J. et al. Opendaylight: Towards a model-driven sdn controller architecture. In: IEEE. *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*. [S.l.], 2014. p. 1–6. Citado na página 75.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011. Citado 2 vezes nas páginas 45 e 47.

MILOJIČIĆ, D.; LLORENTE, I. M.; MONTERO, R. S. Opennebula: A cloud management tool. *IEEE Internet Computing*, IEEE, v. 15, n. 2, p. 11–14, 2011. Citado na página 42.

MININET. *An Instant Virtual Network on your Laptop*. 2014. <http://mininet.org>. [Online; accessed 13-September-2017]. Citado na página 92.

MORREALE, P. A.; ANDERSON, J. M. *Software Defined Networking: Design and Deployment*. [S.l.]: CRC Press, 2015. Citado 2 vezes nas páginas 21 e 27.

MOUGY, A. E. On the integration of software-defined and information-centric networking paradigms. In: IEEE. *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*. [S.l.], 2015. p. 105–110. Citado 4 vezes nas páginas 9, 36, 41 e 54.

NADEAU, T. D.; GRAY, K. *SDN: software defined networks*. [S.l.]: "O'Reilly Media, Inc.", 2013. Citado na página 34.

NGINX.ORG. *Nginx Documentation*. 2018. <http://nginx.org/en/docs/>. [Online; accessed 03-April-2018]. Citado na página 103.

NIKBAZM, R.; DASHTBANI, M.; AHMADI, M. Enabling sdn on a special deployment of openstack. In: IEEE. *Computer and Knowledge Engineering (ICCKE), 2015 5th International Conference on*. [S.l.], 2015. p. 337–342. Citado 3 vezes nas páginas 58, 59 e 60.

OLIVEIRA, R. L. S. D. et al. Using mininet for emulation and prototyping software-defined networks. In: IEEE. *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*. [S.l.], 2014. p. 1–6. Citado na página 66.

ONF, O. N. F. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012. Citado 2 vezes nas páginas 21 e 27.

ONF, O. N. F. *ONF Overview* . 2016. <https://www.opennetworking.org/about/onf-overview/>. [Online; accessed 16-October-2016]. Citado 3 vezes nas páginas 9, 27 e 33.

OPENSTACK.ORG. *What is the openstack?* 2016. <https://www.openstack.org/software/>. [Online; accessed 24-October-2016]. Citado na página 42.

OSSOWSKI, A. *Cache Hit Ratio and Cache Optimization - How to Classify Your CDNs Cache Hit Ratio and Correlate Metrics*. 2015. <https://www.maxcdn.com/blog/cdn-cache-hit-ratio/>. [Online; accessado em 18-Abril-2018]. Citado na página 126.

PFAFF, B. et al. Extending networking into the virtualization layer. In: *Hotnets*. [S.l.: s.n.], 2009. Citado na página 66.

PORTNOY, M. *Virtualization essentials*. [S.l.]: John Wiley & Sons, 2012. Citado na página 64.

POSTEL, J. et al. Rfc 792: Internet control message protocol. *InterNet Network Working Group*, 1981. Citado na página 92.

PURSUIT. *The PURSUIT Project*. 2016. <http://www.fp7-pursuit.eu/PursuitWeb/>. [Online; accessed 28-October-2016]. Citado 3 vezes nas páginas 55, 58 e 60.

PYTHON.ORG. *Documentation: io — Core tools for working with streams*. 2018. <https://docs.python.org/3.5/library/io.html>. [Online; accessed 10-April-2018]. Citado na página 112.

RADEZ, D. *OpenStack Essentials*. [S.l.]: Packt Publishing Ltd, 2015. Citado na página 42.

RAHMEL, D. Testing a site with apachebench, jmeter, and selenium. In: *Advanced Joomla!* [S.l.]: Springer, 2013. p. 211–247. Citado 2 vezes nas páginas 89 e 116.

REN, F. et al. Mobility management scheme based on software defined controller for content-centric networking. In: IEEE. *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*. [S.l.], 2016. p. 193–198. Citado 2 vezes nas páginas 54 e 56.

RHODES, B. et al. *Foundations of Python network programming*. [S.l.]: Springer, 2014. Citado na página 83.

ROCHA, B. *Usando o Flask Cache*. 2014. <http://brunorocha.org/python/flask/usando-o-flask-cache.html>. [Online; accessed 01-March-2018]. Citado na página 82.

ROSSUM, G. V.; DRAKE, F. L. *The python language reference manual*. [S.l.]: Network Theory Ltd., 2011. Citado na página 75.

RYU, S. *Framework*. 2016. Citado na página 99.

RYU, S. *Component Based Software Defined Networking Framework*. 2017. <http://osrg.github.io/ryu/>. [Online; accessed 13-September-2017]. Citado 2 vezes nas páginas 92 e 97.

SAEED, A. et al. Carousel: Scalable traffic shaping at end hosts. In: ACM. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. [S.l.], 2017. p. 404–417. Citado na página 69.

SATO, A. *Instalação e Configuração Nginx*. 2016. <http://stato.blog.br/wordpress/instalacao-e-configuracao-nginx/>. [Online; accessed 02-April-2018]. Citado na página 102.

SCHONWALDER, J.; QUITTEK, J.; KAPPLER, C. Building distributed management applications with the ietf script mib. *IEEE Journal on Selected Areas in Communications*, IEEE, v. 18, n. 5, p. 702–714, 2000. Citado 3 vezes nas páginas 10, 84 e 86.

SHAILENDRA, S. et al. A novel overlay architecture for information centric networking. In: IEEE. *Communications (NCC), 2015 Twenty First National Conference on*. [S.l.], 2015. p. 1–6. Citado na página 36.

SHAMUGAM, V. et al. Software defined networking challenges and future direction: A case study of implementing sdn features on openstack private cloud. In: IOP PUBLISHING. *IOP Conference Series: Materials Science and Engineering*. [S.l.], 2016. v. 121, n. 1, p. 012003. Citado 3 vezes nas páginas 54, 58 e 60.

SINGH, S.; JHA, R. K. A survey on software defined networking: Architecture for next generation network. *Journal of Network and Systems Management*, Springer, v. 25, n. 2, p. 321–374, 2017. Citado na página 90.

SMITH, K. D. et al. Pep-318-decorators for functions and methods. *URL https://www. python. org/dev/peps/pep-0318*, 2003. Citado na página 110.

SON, J. et al. Forwarding strategy on sdn-based content centric network for efficient content delivery. In: IEEE. *2016 International Conference on Information Networking (ICOIN)*. [S.l.], 2016. p. 220–225. Citado 4 vezes nas páginas 23, 54, 56 e 60.

SOOD, M. et al. A survey on issues of concern in software defined networks. In: IEEE. *2015 Third International Conference on Image Information Processing (ICIIP)*. [S.l.], 2015. p. 295–300. Citado na página 30.

STALLINGS, W. *Foundations of modern networking: SDN, NFV, QoE, IoT, and Cloud*. [S.l.]: Addison-Wesley Professional, 2015. Citado 7 vezes nas páginas 9, 11, 32, 43, 44, 46 e 47.

SUN, Q. et al. Sdn-based autonomic ccn traffic management. In: IEEE. *2014 IEEE Globecom Workshops (GC Wkshps)*. [S.l.], 2014. p. 183–187. Citado 4 vezes nas páginas 54, 57, 59 e 60.

SUN, Y.-B.; ZHANG, Y.; ZHANG, H.-L. Survey of research on information-centric networking architecture. *Tien Tzu Hsueh Pao/Acta Electronica Sinica*, 2016. Citado na página 37.

TOMONORI, F. Introduction to ryu sdn framework. *Open Networking Summit*, 2013. Citado na página 74.

TRAJANO, A. F.; FERNANDEZ, M. P. Contentsdn: A content-based transparent proxy architecture in software-defined networking. In: IEEE. *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*. [S.l.], 2016. p. 532–539. Citado 6 vezes nas páginas 24, 54, 55, 56, 58 e 60.

TROSSEN, D.; PARISIS, G. Designing and realizing an information-centric internet. *IEEE Communications Magazine*, IEEE, v. 50, n. 7, 2012. Citado na página 37.

VAHLENKAMP, M. et al. Enabling icn in ip networks using sdn. In: IEEE. *2013 21st IEEE International Conference on Network Protocols (ICNP)*. [S.l.], 2013. p. 1–2. Citado 4 vezes nas páginas 54, 55, 58 e 60.

WANG, X. et al. Livecloud: A lucid orchestrator for cloud datacenters. In: IEEE. *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. [S.l.], 2012. p. 341–348. Citado na página 42.

WAZLAWICK, R. *Metodologia de Pesquisa para Ciência da Computação, 2ª Edição*. [S.l.]: Elsevier Brasil, 2014. Citado na página 49.

XIULEI, W. et al. Sdicn: A software defined deployable framework of information centric networking. *China Communications*, IEEE, v. 13, n. 3, p. 53–65, 2016. Citado 4 vezes nas páginas 54, 55, 60 e 116.

XYLOMENOS, G. et al. A survey of information-centric networking research. *IEEE Communications Surveys & Tutorials*, IEEE, v. 16, n. 2, p. 1024–1049, 2014. Citado 2 vezes nas páginas 37 e 39.

ZHANG, Y.; LI, B. A novel software defined networking framework for cloud environments. In: IEEE. *Cyber Security and Cloud Computing (CSCloud), 2016 IEEE 3rd International Conference on*. [S.l.], 2016. p. 30–35. Citado 5 vezes nas páginas 48, 54, 58, 59 e 60.

Apêndices

APÊNDICE A – Código Fonte do Protótipo

O código fonte do protótipo apresentado no Capítulo 4 são apresentados neste Apêndice.

A.1 Topologia da Rede Local

```
#!/usr/bin/python
# Programa de Pos Graduacao em Ciencia da Computacao
# Departamento de Computacao  DCOMP
# Universidade Federal de Sergipe  UFS
# Erick B Nascimento
# Tree Topology v10  BETA
# Data 25092017

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

    NODE2_IP="172.16.0.23"
    CONTROLLER_IP="172.16.0.23"

    net = Mininet( topo=None,
                   build=False,
                   ipBase="10.0.0.0/8")

    info( "*** Adding controller\n" )
    c0=net.addController(name="c0",
                      controller=RemoteController,
                      ip=CONTROLLER_IP,
                      port=6653)

    info( "*** Add switches\n")
    s1 = net.addSwitch("s1", cls=OVSKernelSwitch)
    s2 = net.addSwitch("s2", cls=OVSKernelSwitch)
    s3 = net.addSwitch("s3", cls=OVSKernelSwitch)
    s4 = net.addSwitch("s4", cls=OVSKernelSwitch)
    s5 = net.addSwitch("s5", cls=OVSKernelSwitch)

    info( "*** Add hosts\n")
    h1 = net.addHost("h1", cls=Host, ip="10.0.0.1")
    h2 = net.addHost("h2", cls=Host, ip="10.0.0.2")
```

```
h3 = net.addHost("h3", cls=Host, ip="10.0.0.3")
h4 = net.addHost("h4", cls=Host, ip="10.0.0.4")
h5 = net.addHost("h5", cls=Host, ip="10.0.0.5")
h6 = net.addHost("h6", cls=Host, ip="10.0.0.6")
h7 = net.addHost("h7", cls=Host, ip="10.0.0.7")
h8 = net.addHost("h8", cls=Host, ip="10.0.0.8")

#Switches
info( "*** Add links 1 GPBS SWs\n")
net.addLink(s1, s2)
net.addLink(s2, s3)
net.addLink(s3, s4)
net.addLink(s4, s5)

info( "*** Add links 10 GPBS Clients \n")
#Clients
h1s2 = {"bw":10000,"delay":"2","loss":0}
net.addLink(h1, s2, cls=TCLink , **h1s2)

h2s2 = {"bw":10000,"delay":"2","loss":0}
net.addLink(h2, s2, cls=TCLink , **h2s2)

h3s5 = {"bw":10000,"delay":"2","loss":0}
net.addLink(h3, s5, cls=TCLink , **h3s5)

h4s5 = {"bw":10000,"delay":"2","loss":0}
net.addLink(h4, s5, cls=TCLink , **h4s5)

h5s3 = {"bw":10000,"delay":"2","loss":0}
net.addLink(h5, s3, cls=TCLink , **h5s3)

h6s1 = {"bw":10000,"delay":"2","loss":0}
net.addLink(h6, s1, cls=TCLink , **h6s1)

h7s3= {"bw":10000,"delay":"2","loss":0}
net.addLink(h7, s3, cls=TCLink , **h7s3)

h8s4 = {"bw":10000,"delay":"2","loss":0}
net.addLink(h8, s4, cls=TCLink , **h8s4)

info( "*** Starting network\n")
net.build()

h1.cmd("ifconfig h1-eth0 mtu 1400 up")
h2.cmd("ifconfig h2-eth0 mtu 1400 up")
h3.cmd("ifconfig h3-eth0 mtu 1400 up")
h4.cmd("ifconfig h4-eth0 mtu 1400 up")
h5.cmd("ifconfig h5-eth0 mtu 1400 up")
h6.cmd("ifconfig h6-eth0 mtu 1400 up")
h7.cmd("ifconfig h7-eth0 mtu 1400 up")
h8.cmd("ifconfig h8-eth0 mtu 1400 up")

info("*** Start NGINX and KeepAlived...[OK]\n")
h6.cmd("service nginx restart")
h6.cmd("service keepalived restart")

info( "*** Starting controllers\n")
for controller in net.controllers:
    controller.start()
```

```
    info( "*** Starting switches\n")
    net.get("s1").start([c0])
    net.get("s2").start([c0])
    net.get("s3").start([c0])
    net.get("s4").start([c0])
    net.get("s5").start([c0])

    info("Create GRE Tunnel\n")
    s1.cmd("ovs-vsctl add-port s1 s1-gre1 -- set interface \
    s1-gre1 type=gre options:remote_ip="+NODE2_IP)

    info( "*** Post configure switches and hosts\n")
    CLI(net)
    net.stop()
if __name__ == "__main__":
    setLogLevel( "info" )
    myNetwork()
```

Código A.1 – Topologia em Árvore

A.2 Topologia da Rede da Nuvem

```
#!/usr/bin/python
# Programa de Pos Graduacao em Ciencia da Computacao
# Departamento de Computacao  DCOMP
# Universidade Federal de Sergipe  UFS
# Erick B Nascimento
# Simple Cloud Topology v 2.1 Stable - GRE Tunel
# Data 25092017
# Last Update 08122017 Description: Add HOST H13

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

    NODE1_IP="172.16.0.22"
    CONTROLLER_IP="127.0.0.1"

    net = Mininet( topo=None,
                   build=False
                   )

    info( "*** Adding Cloud controller\n" )
    c0=net.addController(name="c0",
                         controller=Controller,
                         port=6653
                         )

    info( "*** Add switches\n")
    s8 = net.addSwitch("s8", cls=OVSKernelSwitch)
```

```
    s9 = net.addSwitch("s9", cls=OVSKernelSwitch)

    info( "*** Add links 10 GPBS Clients \n")
    #Clients
    info( "*** Add hosts\n")
    h9 = net.addHost("h9", ip="10.0.0.9")
    h10 = net.addHost("h10", ip="10.0.0.10")
    h11 = net.addHost("h11", ip="10.0.0.11")
    h12 = net.addHost("h12", ip="10.0.0.12")
    h13 = net.addHost("h13", ip="10.0.0.13")

    info( "*** Add links\n")

    s8s9 = {"bw":100,"delay":"2","loss":0}
    net.addLink(s8, s9, cls=TCLink , **s8s9)

    #Reverse Proxy
    net.addLink(h9, s9)

    #Servers
    net.addLink(h10, s9)
    net.addLink(h11, s9)
    net.addLink(h12, s9)
    net.addLink(h13, s9)
    info( "*** Starting network\n")
    net.build()

    h9.cmd("ifconfig h9-eth0 mtu 1400 up")
    h10.cmd("ifconfig h10-eth0 mtu 1400 up")
    h11.cmd("ifconfig h11-eth0 mtu 1400 up")
    h12.cmd("ifconfig h12-eth0 mtu 1400 up")
    h13.cmd("ifconfig h13-eth0 mtu 1400 up")

    info("*** Start NGINX and KeepAlived...[OK]\n")
    h9.cmd("service nginx restart")
    h9.cmd("service keepalived restart")

    info( "*** Starting controllers\n")

    for controller in net.controllers:
        controller.start()

    info( "*** Starting switches\n")
    net.get("s8").start([c0])
    net.get("s9").start([c0])

    info("*** Build GRE Tunel\n")
    s8.cmd("ovs-vsctl add-port s8 s8-gre1 -- set interface \
    s8-gre1 type=gre options:remote_ip="+NODE1_IP)


    info( "*** Post configure switches and hosts\n")

    CLI(net)
    net.stop()

if __name__ == "__main__":
    setLogLevel( "info" )
```

```
    myNetwork()
```

Código A.2 – Topologia Linear

A.3 Código Fonte do Microwebserver

```
#!/usr/bin/python
# Web Server RESTful Services
# Erick B. Nascimento
# Universiade Federal de Sergipe - UFS
# PROCC-DCOMP
# V1.2
# Dara: 18/09/2017
# Build: Stable

# SERVER01
from flask import Flask, abort, send_file
from flask import jsonify
from flask import request
from io import BytesIO
import re
from flask import render_template, url_for, redirect
from flask_caching import Cache
import io

cache = Cache(config={'CACHE_TYPE':'filesystem',\
'CACHE_DIR':'/home/mininet/SERVER01/cachesrv1'})
app = Flask(__name__)
cache.init_app(app)

@app.route("/")
def index():
        message = "Server 01 - RUN"
        return render_template("index.html", message=message)


#Methods IMAGES
@app.route("/images")
def images():
        return render_template("images.html")


@app.route("/static/images/125" ,methods=["GET"])
@cache.cached(timeout=30)
def staticImages125():
        with open("static/images/125.jpg", 'rb') as bites:
                return send_file(
                        io.BytesIO(bites.read()),
                        attachment_filename='125.jpg',
                        mimetype='image/jpg'
                )
@app.route("/static/images/250" ,methods=["GET"])
@cache.cached(timeout=30)
def staticImages250():
        with open("static/images/250.jpg", 'rb') as bites:
                return send_file(
                        io.BytesIO(bites.read()),
                        attachment_filename='250.jpg',
```

```
                        mimetype='image/jpg'
                )
@app.route("/static/images/500" ,methods=["GET"])
@cache.cached(timeout=30)
def staticImages500():
        with open("static/images/500.jpg", 'rb') as bites:
                return send_file(
                        io.BytesIO(bites.read()),
                        attachment_filename='500.jpg',
                        mimetype='image/jpg'
                )
@app.route("/static/images/1000" ,methods=["GET"])
@cache.cached(timeout=30)
def staticImages1000():
        with open("static/images/1000.jpg", 'rb') as bites:
                return send_file(
                        io.BytesIO(bites.read()),
                        attachment_filename='1000.jpg',
                        mimetype='image/jpg'
                )
# Main Method
if __name__ == "__main__":
        app.run(debug=True, host = "", port = 80)
```

Código A.3 – MicroWebServer Flask

APÊNDICE B – Arquivos de Configuração dos *Daemons* e Relatórios

B.1 Configuração da Resolução de Nomes (*Host*)

```
127.0.0.1       localhost
127.0.1.1       mininet-vm01
172.16.0.22     mininet-vm01.domain.local       mininet-vm01

#MININET SERVERS SDN LAN/CLOUD
10.0.0.1        srvweb01.domain.local   srvweb01
10.0.0.2        srvweb02.domain.local   srvweb02
10.0.0.5        srvweb05.domain.local   srvweb05
10.0.0.7        srvweb07.domain.local   srvweb07
10.0.0.10       srvweb10.domain.local   srvweb10
10.0.0.11       srvweb11.domain.local   srvweb11
10.0.0.12       srvweb12.domain.local   srvweb12
10.0.0.13       srvweb13.domain.local   srvweb13

#LOAD BALANCE
10.0.0.6        srvlb01.domain.local    srvlb01
10.0.0.9        srvlb02.domain.local    srvlb02

#WEBSITE
10.0.0.6        example.domain.local    example.com
10.0.0.9        example.domain.local    example.com

#LOAD SERVERS VMs

172.16.0.23     mininet-vm02.domain.local       mininet-vm02
172.16.0.24     mininet-vm03.domain.local       mininet-vm03
```

Código B.1 – Resolução de Nomes

B.2 Configuração do Proxy Reverso

```
upstream slicelan {
        least_conn;
        server srvweb01.domain.local:80;
        server srvweb02.domain.local:80;
        server srvweb05.domain.local:80;
        server srvweb07.domain.local:80;
        server srvweb10.domain.local:80;
        server srvweb11.domain.local:80;
        server srvweb12.domain.local:80;
        server srvweb13.domain.local:80;

}

server {
```

```
    listen 80;
        access_log off;
        expires max;
        location / {
                client_max_body_size    1m;
                client_body_buffer_size 128k;
                proxy_send_timeout   90;
                proxy_read_timeout   90;
                proxy_buffer_size    2k;
                proxy_buffers     16 4k;
                proxy_connect_timeout 30s;
                proxy_pass http://slicelan;
                proxy_set_header   Host   $host;
                proxy_set_header   X-Real-IP  $remote_addr;
                proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
        }
        location /static/images {
                proxy_cache cachecloud;
                client_max_body_size    1m;
                client_body_buffer_size 128k;
                proxy_send_timeout   90;
                proxy_read_timeout   90;
                proxy_buffer_size    2k;
                proxy_buffers     16 4k;
                proxy_connect_timeout 30s;
            proxy_pass http://slicelan;
                proxy_set_header   Host   $host;
                proxy_set_header   X-Real-IP  $remote_addr;
                proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
        }
}
```

Código B.2 – Proxy Reverso

B.2.1 Zona de Cache

```
worker_processes 4;
pid /run/nginx.pid;
events {
        worker_connections 768;
        # multi_accept on;
}
##########   ZONAS DE CACHE NA LAN ##########
http {
        #Zona de Cache Fisico
        proxy_cache_path /home/mininet/CACHECLOUD/ \
        keys_zone=cachecloud:10m max_size=1m inactive=1m;

        ##
        # Basic Settings
        ##
        sendfile on;
        tcp_nopush on;
        tcp_nodelay on;
        keepalive_timeout 65;
        types_hash_max_size 2048;
        include /etc/nginx/mime.types;
        default_type application/octet-stream;
```

```
        ##
        # Logging Settings
        ##
        access_log /var/log/nginx/access.log;
        error_log /var/log/nginx/error.log;

        ##
        # Gzip Settings
        ##
        gzip on;
        gzip_disable "msie6";
        gzip_buffers 16 8k;

        ##
        # Virtual Host Configs
        ##
        include /etc/nginx/conf.d/*.conf;
        include /etc/nginx/sites-enabled/*;
}
```

Código B.3 – Cache Físico

B.3 Configuração do Balanceamento de Carga

```
#/bin/bash
global_defs {
lvs_id_nginx_DH
}

vrrp_script_check_nginx {
script "killall -0 nginx"
script "/etc/init.d/keepalived/check_nginx_status.sh"

interval 2
weight 2
}

vrrp_instance VI_1 {
    state MASTER
    interface h6-eth0
    virtual_router_id 101
    priority 101
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        10.0.0.100
    }

track_script{
        check_nginx
        }
}
```

Código B.4 – Configuração do LB

Anexos

ANEXO A – Sistema Operacional da Rede (*Network Operating System – NOS*)

A.1 Sistema Operacional tipo *Learning Switch*

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types

class OpenVSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(OpenVSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
    CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
        actions = [parser.,
        OFPActionOutput(ofproto.OFPP_CONTROLLER,
                        ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority,\
    match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(
        ofproto.OFPIT_APPLY_ACTIONS,actions)]

        if buffer_id:
            mod = parser.OFPFlowMod(
            datapath=datapath, buffer_id=buffer_id,
            priority=priority, match=match,
            instructions=inst)
        else:
            mod = parser.OFPFlowMod(
            datapath=datapath, priority=priority,
            match=match, instructions=inst)
        datapath.send_msg(mod)
```

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):

    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug(
        "packet truncated: only %s of %s bytes",
         ev.msg.msg_len, ev.msg.total_len)
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:

        return
    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", \
            dpid, src, dst, in_port)


    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port,\
         eth_dst=dst)

        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 1, \
            match, actions, msg.buffer_id)
            return
        else:
            self.add_flow(datapath, 1, match, actions)
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(
    datapath=datapath, buffer_id=msg.buffer_id,
    in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

Código A.1 – Sistema Operacional